Efficient Processing of Skyline Queries on Static Data Sources, Data Streams and Incomplete Datasets

by

Mithila Nagendra

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved August 2014 by the
Graduate Supervisory Committee:

Kasim Selçuk Candan, Chair
Yi Chen
Hasan Davulcu
Yasin N. Silva
Hari Sundaram

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

Skyline queries extract *interesting* points that are non-dominated and help paint the *bigger picture* of the data in question. They are valuable in many multi-criteria decision applications and are becoming a staple of decision support systems.

An assumption commonly made by many skyline algorithms is that a skyline query is applied to a single static data source or data stream. Unfortunately, this assumption does not hold in many applications in which a skyline query may involve attributes belonging to multiple data sources and requires a *join* operation to be performed before the skyline can be produced. Recently, various *skyline-join* algorithms have been proposed to address this problem in the context of static data sources. However, these algorithms suffer from several drawbacks: they often need to scan the data sources exhaustively to obtain the skyline-join results; moreover, the pruning techniques employed to eliminate tuples are largely based on expensive tuple-to-tuple comparisons. On the other hand, most data stream techniques focus on single stream skyline queries, thus rendering them unsuitable for *skyline-join* queries.

Another assumption typically made by most of the earlier skyline algorithms is that the data is complete and all skyline attribute values are available. Due to this constraint, these algorithms cannot be applied to incomplete data sources in which some of the attribute values are missing and are represented by NULL values. There exists a definition of dominance for incomplete data, but this leads to undesirable consequences such as *non-transitive* and *cyclic* dominance relations both of which are detrimental to skyline processing.

Based on the aforementioned observations, the main goal of the research described in this dissertation is the design and development of a framework of skyline operators that effectively handles three distinct types of skyline queries: 1) *skyline-join*

queries on static data sources, 2) *skyline-window-join* queries over data streams, and 3) *strata-skyline* queries on incomplete datasets. This dissertation presents the unique challenges posed by these skyline queries and addresses the shortcomings of current skyline techniques by proposing efficient methods to tackle the added overhead in processing skyline queries on static data sources, data streams, and incomplete datasets.

DEDICATION

To my dear dad

You will always be missed

ACKNOWLEDGEMENTS

I take this opportunity to extend my heartfelt gratitude to my advisor and mentor, Prof. K. Selçuk Candan. It would have been impossible for me to reach this milestone without his invaluable feedback and continued support during the course of the degree. His trust in my capabilities and patience in helping me understand new concepts has been the motivating force behind this work. I am grateful to Dr. Yi Chen, Dr. Hari Sundaram, Dr. Hasan Davulcu, and Dr. Yasin Silva for being a part of my committee. Their valuable inputs on my dissertation helped make it complete and comprehensive.

I would like to express my appreciation to the School of Computing, Informatics, and Decision Systems Engineering at Arizona State University for giving me the opportunity to further my education through the Ph.D. program. I would like to thank my graduate advisors, Araxi Hovhannessian and Cynthia Donahue, for giving me a clear picture of all the rules and regulations of the Graduate College and for their help with all the official documents. I would also like to thank Monica Dugan, Pamela Dunn, and Theresa Chai for all their help over the years.

I would like to thank Xilun, Jung, Xinsheng, Shengyu, Sicong, Sriram, and Yash, my colleagues at the EmitLab, for their insightful discussions. A special thank you goes out to Mijung Kim, for her thoughtful feedback and advice, and Parth Nagarkar, for his positive attitude and undying sense of humor. It was truly a pleasure working with this group of people.

I am very thankful to my husband, Adarsh Narasimhamurthy. His incessant patience, help and love largely contributed towards this accomplishment. I would also like to thank Adarsh's parents, Narasimhamurthy and Radha Murthy, and his brother, Arvind Narasimhamurthy, for their love and encouragement.

Last but not least, I am eternally grateful to my father, Nagendra Veeraiah.

His inspirational words of wisdom will never be forgotten. I am extremely thankful to my mother, Vasundhara Nagendra, sister, Megha Nagendra Wells, and brother-in-law, David John Wells. Without their love, strength and constant support this achievement would have been quite impossible.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

There has been a growing interest in the area of skyline query research. Intuitively, skyline queries extract *interesting* points that help paint the "bigger picture" of the data in question, providing insight into the diversity of the data across different features. Given a set, D, of data points in a feature space, the *skyline* of D consists of the points that are not dominated[1] by any other data point in D [13].

Searching for non-dominated data is valuable in many applications that involve multi-criteria decision making [65]. Figure 1.1 illustrates one such application that finds the skyline of late-night restaurants. This skyline application might be useful to students in a university who stay up late at night and need a snack at odd hours. Figure 1.1 shows the ratings and closing times of a set of restaurants: the points that are connected represent restaurants that are part of the skyline; this includes highest-rated restaurants that are open late into the night. Other restaurants are not part of the skyline because they are dominated in terms of time and/or rating by at least one restaurant that is in the skyline. The shaded area in Figure 1.1 is the dominance region of restaurant b: for any restaurant in this range, restaurant b is either open till a later time and/or has a better rating. Therefore, b is said to be more *interesting* than all restaurants it dominates.

Skyline Queries are becoming a staple of decision support systems. As a result, the task of processing skyline queries in an efficient manner has attracted considerable attention. Way back in 1975, Kung *et al.* studied the problem of finding non-

[1] A point dominates another point if it is as good or better in all attributes, and better in at least one attribute.

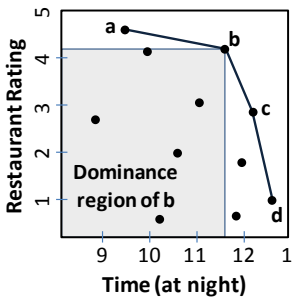


Figure 1.1: Skyline of late-night restaurants

dominated data under the name of the maximum vector problem [38]. Later, Borzsonyi *et al.* [13] coined the term *skyline* and investigated skyline queries in the context of databases. Since then, a plethora of skyline algorithms have been designed for various scenarios. These include sort-based techniques [15, 9], progressive methods [70], online algorithms [37, 55, 39] and algorithms for high dimensional datasets [50]. Also, there has been research efforts in developing skyline algorithms for other environments, such as skyline-join processing [36, 68, 78], data streams [44, 71, 87], imprecise datasets [57, 3, 45, 34] and parallel environments [67, 84, 77, 28].

1.1 Shortcomings of Existing Techniques

A particular shortcoming of many of the existing skyline algorithms (for example, [38, 13, 39, 44, 71]) is that they primarily focus on single-source skyline processing in which all required skyline attributes are present in the same source. In other words, these algorithms commonly make an assumption that a skyline query is applied to a single static data source or data stream. However, this assumption does not hold true in many applications that require integration of data from different sources. In such scenarios, a skyline query may involve attributes belonging to multiple data sources, thus making the *join* operation an integral part of the overall process. For instance, in static environments integrated *skyline-join* queries maybe necessary over complex schemas in which the data is distributed onto many sources, whereas in stream en-

vironments such integration maybe needed for streams that originate from different sensors or from multiple sources in a distributed publish/subscribe architecture. Going back to our earlier example (Figure 1.1), in addition to the time and restaurant rating attributes, students might also consider the distance of a restaurant to the university to be a factor in their decision-making process. If this information is available from a different source, we would then need to join the relevant sources in order to obtain the restaurants that are part of the skyline.

Recently, various *skyline-join* algorithms (for instance, [36, 68, 78]) have been proposed to address this problem in the context of static data sources. However, these algorithms suffer from several drawbacks. They often need to scan the data sources exhaustively in order to obtain the skyline-join results. Moreover, the pruning techniques employed to eliminate the tuples are largely based on time-consuming tuple-to-tuple comparisons, instead of being block-based.

In the context of data streams, several algorithms (such as [44, 71]) have been developed to continuously monitor the changes in the skyline based on the arrival of new tuples and expiration of old ones. Unfortunately, most of these techniques focus on skyline queries in which the skyline attributes belong to a single data stream, thus rendering them unsuitable for *skyline-join* queries that require a real-time *join* operation to be carried out during skyline query processing.

Another shortcoming of most of the earlier skyline algorithms (like [13, 37, 15, 55, 39] is that they count on the data being precise. In particular, these techniques typically make an assumption that the data is complete and all skyline attribute values are available. However, this assumption is not valid in many applications that involve incomplete data sources in which some of the attribute values are missing (and are represented by NULL values) due to reasons like data-entry errors, lack of knowledge, and privacy. For example, if we consider a set of movies rated by different

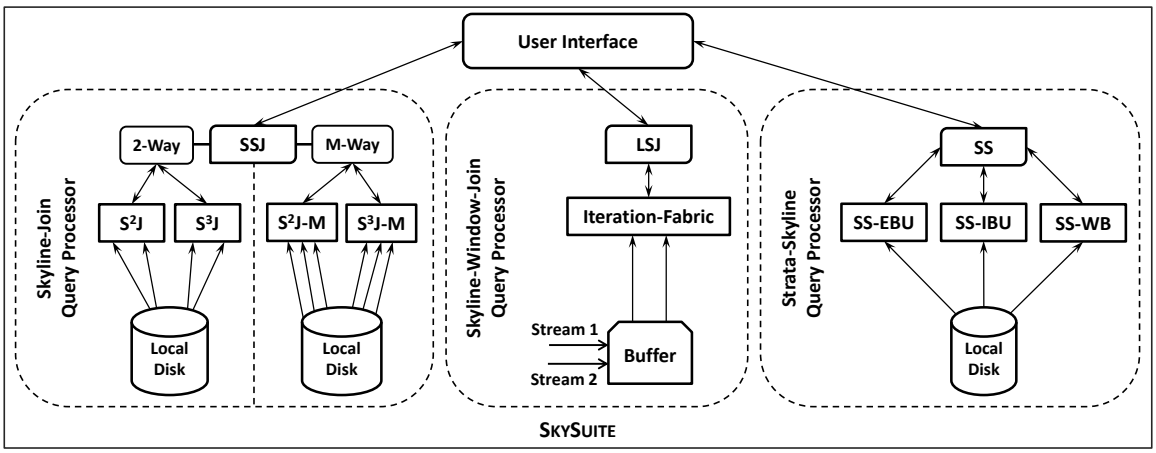


Figure 1.2: `SkySuite`: a framework of skyline operators that effectively handles *skyline-join* queries on static data sources, *skyline-window-join* queries over data streams, and *strata-skyline* queries on incomplete data sources

users, many movie ratings may be missing as most often users do not rate movies that they have not seen.

Missing values can complicate the definition of skylines and lead to extra overheads in skyline processing [2]. There exists a definition of dominance for incomplete data [34], however, this leads to undesirable consequences and counter-intuitive results. These include *non-transitive* and *cyclic* dominance relationships, both of which are incompatible with any intuitive interpretation of skylines and are also detrimental to efficient skyline processing. For instance, cyclic dominance can lead to a scenario in which the skyline set is empty, whereas loss of transitivity renders useless many of the existing optimization techniques like indexing and data pruning.

Based on the above observations, the main goal of the research described in this dissertation is the design and development of a framework of skyline operators, named `SkySuite` (Figure 1.2), that effectively handles three distinct types of skyline queries: 1) *skyline-join* queries on static data sources, 2) *skyline-window-join* queries over data streams, and 3) *strata-skyline* queries on incomplete data sources. This dissertation presents the unique challenges posed by these skyline queries and proposes novel techniques to address the added overhead in processing skyline queries on static

data sources, data streams, and incomplete datasets. The following section gives an overview of the key contributions presented in this dissertation.

1.2 Research Contributions

The objective of the research work elucidated in this dissertation is the architecture and building of the key components of `SkySuite`: a framework of skyline operators that effectively handles *skyline-join* queries on static data sources, *skyline-window-join* queries over data streams, and *strata-skyline* queries on incomplete data sources. In particular, as shown in Figure 1.2, the `SkySuite` framework: a) effectively processes Two-way (2-way) and Multi-way (M-way) *skyline-join* queries on static data sources by leveraging the novel `S`kyline-`S`ensitive `J`oin (`SSJ`) operator, b) incrementally maintains *skyline-window-join* results over pairs of data streams by utilizing the `L`ayered `S`kyline-window-`J`oin (`LSJ`) operator, and c) efficiently executes *strata-skyline* queries on incomplete data sources by taking advantage of the unique `S`trata-`S`kyline (`SS`) operator. This dissertation will illustrate the key methodologies behind the `SSJ`, `LSJ` and `SS` operators. The following sections summarize the contributions of the research work presented in this dissertation.

1.2.1 Skyline-Sensitive-Join (SSJ) Operator for Processing Skyline-Join Queries on Static Data Sources

At the core of the `S`kyline-`S`ensitive `J`oin (`SSJ`) operator are two novel *skyline-join* algorithms, namely *Skyline-Sensitive Join* ($\texttt{S}^2\texttt{J}$) and *Symmetric Skyline-Sensitive Join* ($\texttt{S}^3\texttt{J}$), that process skyline-join queries over pairs of static data sources. The proposed approaches compute skyline-join results using a novel *Layer/Region pruning* (*LR-pruning*) technique that prunes the join space in terms of blocks of data, as opposed to individual data points, thereby avoiding excessive time-consuming pair-

wise point-to-point dominance checks. Furthermore, the S^{3}J algorithm utilizes an early stopping condition in order to successfully compute the skyline-join results by accessing only a subset of the input tables.

In addition to S^{2}J and S^{3}J, the SSJ operator also leverages the S^{2}J-M and S^{3}J-M algorithms. These algorithms extend S^{2}J's and S^{3}J's two-way skyline-join ability to efficiently process skyline-join queries over more than two data sources. S^{2}J-M and S^{3}J-M leverage the extended concept of *LR-pruning*, called M-way *LR-pruning*, to compute multi-way (M-way) skyline-joins in which more than two data sources are integrated during skyline processing.

Extensive experimental results help confirm the advantages of the proposed algorithms over the state-of-the-art skyline-join techniques.

1.2.2 Layered Skyline-window-Join (LSJ) Operator for Executing Skyline-Window-Join Queries on Data Streams

The Layered Skyline-window-Join (LSJ) operator processes *skyline-window-join* queries over pairs of data streams by maintaining skyline-join results over sliding windows in a layered, incremental manner. The LSJ operator is the first of its kind in addressing skyline-window-join queries over data streams. LSJ makes a first attempt in solving the problem of answering skyline-window-join queries by combining the advantages of existing skyline methods (including those that efficiently maintain skyline results over a single stream and those that compute skyline-joins of pairs of static datasets) to develop a novel *iteration-fabric* skyline-window-join processing structure. Using the *iteration-fabric*, LSJ eliminates redundant work across consecutive windows by leveraging shared data across all iteration layers of the windowed skyline-join process. Moreover, it maintains skyline-window-join results in an incremental manner by continuously monitoring the changes in all layers of the skyline-join process.

Extensive experimental evaluations over real and simulated data help show that LSJ provides large gains over naive extensions of existing schemes which are not designed to eliminate redundant work across multiple processing layers.

1.2.3 Strata-Skyline (SS) Operator for Processing Strata-Skyline Queries on Incomplete Datasets

At the heart of the Strata-Skyline (SS) operator are three unique *strata-skyline* algorithms, namely Strata-Skyline-using-Explicit-Bitmaps-for-Unknowns (SS-EBU), Strata-Skyline-using-Implicit-Bitmaps-for-Unknowns (SS-IBU) and Strata-Skyline-using-Weighted-Bitmaps (SS), that rely on novel definitions of *potential dominance* in the presence of NULL values to efficiently process strata-skyline queries over incomplete data sources. The proposed SS operator does not suffer from undesirable effects such as *non-transitive* and *cyclic* dominance relationships, both of which are detrimental to skyline processing. The SS operator does not produce a single set of skyline results, but instead utilizes the proposed dominance definitions and special bitmap indices to stratify the tuples in an incomplete data source into *strata* (or *layers*) of varying degrees of *skyline potential*.

Extensive experimental evaluations help confirm the advantages of the SS operator over naive extensions of existing schemes which are not designed to handle the added overhead in processing skyline queries over incomplete data sources.

1.3 Dissertation Overview

The rest of the dissertation is structured as follows:

- Chapter 2 presents an overview of the existing work in the field of skyline query processing over static data sources, data streams and incomplete datasets.

- Chapter 3 gives a detailed explanation of the Skyline-Sensitive Join (SSJ) operator and the related algorithms.
- Chapter 4 provides a comprehensive description of the ideas and methodologies behind the Layered Skyline-window-Join (LSJ) operator.
- Chapter 5 elucidates the key concepts and algorithms leveraged by the Strata-Skyline (SS) operator.
- Lastly, Chapter 6 concludes this dissertation.

Chapter 2

RELATED WORK

This chapter provides an overview of the existing work in the fields of skyline query processing and top-k join processing. The literature on top-k join query processing is relevant to this research as top-k joins are used as part of the multi-way skyline-join framework. Therefore, an overview of the related literature on top-k join query processing is also provided here.

2.1 Skyline Algorithms

The task of finding the non-dominated set of data points was attempted by Kung *et al.* in 1975 under the name of the maximum vector problem [38]. The algorithm proposed in [38] is quite complex and is based on the divide and conquer principle. Kung's algorithm lead to the development of various skyline algorithms designed for specific situations, including those devised for high dimensional data sets [50] and parallel environments [67]. It also inspired novel skyline algorithms for static [13, 78] and stream environments [44, 71].

Borzsonyi *et al.* [13] were the first to coin and investigate the *skyline* computation problem in the context of databases. The authors extended Kung's divide and conquer algorithm so that it works well on large databases. In particular, they proposed a *Block-Nested-Loops (BNL)* algorithm, which keeps an in-memory window of incomparable points and reports a point as a skyline result only if it is not dominated by any other point in the database. They also proposed a *divide and conquer* based algorithm, which divides the data space into several regions, calculates the skyline in each region, and produces the final skyline from the points in the regional skylines.

By focusing on numerical domains, Borzsonyi *et al.* were able to gain logarithmic complexity along the lines of work done in [38].

The *Sort-Filter-Skyline (SFS)* algorithm [15], which is based on the same principle as the BNL algorithm, improves on performance by first sorting the data according to a monotone function. Bartolini *et al.* also developed a sort-based skyline technique called the *Sort and Limit Skyline algorithm (SaLSa)* that uses the idea of presorting input tuples to limit the number of tuples read and compared during skyline query processing [10]. Tan *et al.* proposed progressive skyline algorithms called *bitmap* and *index* [70]. The *bitmap* method is completely non-blocking and exploits a bitmap structure to quickly identify whether a point is a skyline result or not. The *index* method, on the other hand, exploits a transformation mechanism and a B^+-tree index to return skyline points in batches. Other contributions to skyline query processing include online algorithms, such as [37] and [55], based on nearest-neighbor search.

Lee et al. [39] proposed an index structure called *ZBtree* to index and store data points based on the Z-order curve. They also developed several skyline algorithms that utilize the *ZBtree* index to efficiently process skyline queries. Huang *et al.* [28] also proposed a parallel skyline algorithm for multi-processor clusters that utilizes Z-order clustering to reduce dominance checks. As discussed in Section 3.3.1, we also leverage a Z-order based index structure to help prune the join space during skyline-sensitive join computations.

Recently, Shang *et al.* examined the skyline operator in the context of anti-correlated distributions [62]. The authors proposed a probabilistic cardinality model for anti-correlated distributions and analyzed the upper and lower bounds of the expected value of skyline cardinality. They also developed an algorithm called *SOAD (Skyline Operator on Anti-correlated Distributions)* that effectively eliminates non-promising points based on a *determination* and *elimination* framework.

2.2 Query Processing over Multiple Static Data Sources

One key assumption made by the aforementioned algorithms is that the skyline query is applied to a single table and thus, all the skyline attributes are present in one table. Since this assumption does not hold in many applications in which the skyline attributes are split onto many tables, it lead to the development of algorithms that focus on the efficient processing of skyline queries over multiple data tables. In this section, we first give an overview of the techniques in distributed skyline processing and then describe the literature for skyline-join query processing.

2.2.1 Distributed Skyline Query Processing

Several techniques have been proposed in the past that mainly focus on skyline computation in highly distributed systems, such as P2P systems, in which each server stores a part of the available information (i.e., the input data is stored in a decentralized manner) and is assembled at query time [7, 26, 27]. Wolf *et al.* addressed the problem of skyline queries in web information systems [7]. The authors proposed a technique for processing distributed skyline queries and also developed heuristics to speed up the process of retrieving distributed skyline results. Hose *et al.* [26, 27] gave a comprehensive survey of the existing techniques in distributed skyline query processing. They illustrated that skyline processing in highly distributed environments comes with inherent challenges that require the development of special distributed skyline techniques.

Recently, Trimponias *et al.* developed a framework called *Vertical Partition Skyline (VPS)* for processing skyline queries over a dataset that is vertically decomposed among many servers [74]. The authors proposed a technique that focuses on minimizing the transmission and communication costs between servers during skyline

processing. This method computes skyline results by combining the vertical decompositions in different servers on the primary keys of the vertical splits. As can be observed, this approach is relevant to the skyline-join problem and skyline-join can be viewed as an instance of VPS. But, as we elucidate in the following section, the problem of skyline-join query processing is more general and is applicable even in scenarios where the integration of data sources occurs on any arbitrary join attribute.

2.2.2 Skyline-Join Query Processing

Prior work on skylines over multiple data sources include [32, 68, 78, 36, 60, 59, 12, 41, 33]. Jin *et al.* were the first to coin and study the *skyline-join* problem in the context of multi-relational databases [32]. The authors proposed algorithms that integrate state-of-the-art join methods into skyline computation. Sun *et al.* extended this work by introducing a new criteria to prune the join space and two algorithms to support *skyline-join* queries [68]. The first of these extends the *Sort and Limit Skyline (SaLSa)* algorithm [9] to cope with multiple relations, whereas the second prunes the search space iteratively. These methods suffer from several drawbacks, including multiple passes over the datasets and complex book-keeping to identify pruned tuples.

In [33], the authors developed non-blocking methods for evaluating skylines in the presence of equi-joins. These algorithms are built on top of the traditional *Nested-Loop* and *Sort-Merge* join algorithms. Raghavan *et al.* in [59] proposed a progressive query evaluation framework called *ProgXe* that transforms the execution of queries involving skyline over joins into a non-blocking form. The framework also enables skyline-join queries to be processed in a progressive manner, where the query processing (join, mapping, and skyline) is conducted at multiple levels of abstraction. *ProgXe* exploits knowledge gained from both input as well as mapped output spaces

to enable executions of joins and skylines at a higher granularity of abstraction, rather than at the level of individual tuples.

Following [59], Raghavan *et al.* in [60] proposed a framework called *SKIN (SKyline INside Join)* to evaluate SkyMapJoin queries. SKIN reduces the total number of join results generated and the number of dominance comparisons needed to compute the skyline results by performing query evaluation at various levels of abstraction. In [12], the authors developed various algorithms to efficiently process *aggregate skyline-join* queries. These algorithms locally process skylines as much as possible before carrying out the join between the tuples. [41] proposed a framework called *FlexPref*, which aims to support a wide array of preference methods at the core of a database system. *PrefJoin* [36], which builds on FlexPref, is a preference-aware join query operations designed specifically to deal with preference queries over multiple relations.

Recently, Vlachou *et al.* [78] introduced a *Sort-First-Skyline-Join (SFSJ)* algorithm that fuses the identification of skyline tuples with the computation of the join. SFSJ computes the skyline set by accessing only a subset of the input tuples; it alternates between its inputs and generates the skyline tuples progressively as it computes the join results. The SFSJ algorithm relies on an early-termination condition, applied on a simple model of sorted input access, to determine whether it has accessed enough tuples to generate the complete skyline set. The authors analyze the performance of SFSJ under 2 data pulling strategies, simple round-robin (SFSJ-RR) and a strategy (SFSJ-SC) that adapts the order in which the input relations are accessed. The adaptive strategy prioritizes accesses to the input relations and is shown to be optimal for SFSJ in terms of the number of tuples accessed. SFSJ also provides a way to prune the input tuples if they do not contribute to the set of skyline-join results, thus reducing the number of generated join results and dominance checks.

One key deficiency of SFSJ, which we aim to tackle in this paper, is that it

largely depends on expensive tuple-to-tuple comparisons to find the regions that can be pruned. Our proposed approaches overcome this drawback by pruning the join space in terms of blocks as opposed to individual data points, thereby avoiding time-consuming pairwise point-to-point dominance checks. In addition to this, we introduce an early stopping condition that enables our approach to scan less input tuples than SFSJ and perform as good or better than SFSJ. Our approach is able to make pruning decisions more proactively than SFSJ, thus enabling it to see less input tuples during the skyline-join process. In the following chapter, we discuss the two-way solutions and then extend these ideas to multi-way (M-way) skyline-joins in which more than two data sources are integrated during skyline processing.

2.2.3 Top-k Join Algorithms

Over several decades, a plethora of techniques have been developed for the efficient processing of top-k join queries in various environments [30]. Most of these algorithms are M-way by nature, meaning that they join results from M datasets simultaneously.

Fagin *et al.* were the first to propose efficient top-k join algorithms for middleware environments [22, 23]. In [23], the *Threshold Algorithm (TA)* and the *No Random Access (NRA)* algorithm were introduced. TA assumes that both sorted and random access methods are supported by the data sources. It utilizes random accesses to obtain the overall score of an object soon after it is seen in one of the data sources. On the other hand, NRA computes top-k results by considering only sorted accesses. Ilyas *et al.* developed a *rank-join* algorithm that supports top-k join queries in relational databases [29]. The algorithm is based on the idea of *ripple join* [25] and it incrementally computes top-k join results by scanning the inputs ordered on their scoring predicates. In [53], Natsev *et al.* proposed the J^* algorithm that finds top-k results by maintaining partial and complete join results in a priority queue that is

sorted on the upper bounds of the total scores. Marian *et al.* introduced an efficient algorithm in [49] that minimizes the response time of top-k queries over web-accessible databases. It does so by maximizing the parallelism of source accesses and using the *Upper strategy* in the case where only random access is possible. In [83], Wu *et al.* proposed a branch-and-bound algorithm that finds a sort order of the inputs which minimizes wasted work in the computation of top-k results. More recently, Sahpaski *et al.* proposed an algorithm that answers top-k join queries by translating them into a number of range queries [61]. The range queries are calculated by using histograms on the data distributions of the input attributes.

Readers interested in top-k join queries are encouraged to leveraged the excellent survey by Ilyas and his colleagues on this topic [30].

2.3 Query Processing over Data Stream Environments

Over the last decade, there has been a growth in the number of applications in which data arrives in a streaming manner and at high speeds [5]. These include financial applications that process streams of stock market or credit card transactions, telephone call monitoring applications that process streams of call-detail records [16], network traffic monitoring and sensor network applications that analyse environmental data gathered by sensors [17, 20, 48]. These applications often require long-running, continuous queries as opposed to the traditional one-time queries. Thus, the advent of a wide array of stream-based applications has necessitated a push towards the development of algorithms that take into consideration the constant changes in stream environments.

The following sections provide an overview of the existing work in the fields of skyline, top-k and join query processing over streaming data.

2.3.1 Join Processing and Top-k Queries over Data Streams

Streaming algorithms for join processing are relevant to our research. Continuous join queries over stream environments are needed for correlating data from multiple data streams [4]. [81] proposed a symmetric hash join method that is optimized for in-memory performance. Following this, a plethora of techniques have been developed for efficiently processing join queries over data streams [20, 24, 75, 76, 42, 18, 72, 73, 85, 64, 79, 54].

Many of these works focus on efficiently eliminating join processing redundancies across consecutive time windows to maximize the output rate [76, 42, 85]. Others focus on memory; they present join processing and load shedding techniques that minimize loss in accuracy when the memory is insufficient to process all incoming data [64, 6, 73, 58].

[79] developed a novel *State-Slice* sharing paradigm for window join queries. In particular, it proposed a method to slice window states into fine-grained slices that form a chain of sliced window joins to reduce the number of joins from quadratic to linear. [54] developed an algorithm called *SCUBA* that utilizes dynamic clustering to optimize the execution of multiple continuous queries on spatio-temporal streams. This cluster-based solution helps reduce unnecessary joins and improves query performance on moving objects.

Recent works on data streams also include investigation of richer query semantics such as top-k and skyline. Since the mid 2000s, there has been a plethora of work on processing streaming top-k queries. In an early work, Mouratidis *et al.* addressed the problem of answering continuous top-k queries over a single stream [51]. The authors proposed two algorithms: (a) the *TMA* algorithm, which computes new answers to a query every time some of the current top-k results expire, and (b) the *SMA*

algorithm that reduces top-k queries to k-skyband queries in order to avoid complete re-computation when some results expire. A very recent work in this area includes [63] which presents a framework called *MTopS* that handles multiple continuous top-k queries executed simultaneously against a common data stream. Other works on data streams examine top-k join processing [58, 80]. These primarily focus on maintaining the top-k join results and candidate lists incrementally, as the data streams and their scores evolve over time.

As skyline is more related to the contributions of this work, a detailed discussion of the literature on streaming skyline techniques is presented in the following section.

2.3.2 Skyline Processing over Data Streams

As mentioned earlier (Section 2.1, 2.2.1 and 2.2.2), the problem of skyline query processing has been extensively studied in the conventional setting of static data. There is a large body of work that represents the research carried out in both single-source skyline processing [13, 15, 9] and multiple source skyline-join processing [68, 78]. These methods assume that the data is entirely available and unchanging at the time of query execution, and focus on computing a single skyline rather than continuously tracking skyline changes in an incremental manner.

The advent of a wide array of stream-based applications has necessitated a push towards the development of algorithms that take into consideration the constant changes in stream environments. Recently, several algorithms have been developed to track skyline changes over data streams [44, 71, 69]. These methods are able to continuously monitor the changes in the skyline according to the arrival of new tuples and expiration of old ones.

Sun *et al.* addressed skyline queries over distributed data streams [69], where the streams are derived from multiple horizontally split data sources. The authors

developed an algorithm called *BOCS* that consists of an efficient centralized algorithm, *GridSky*, and an associated communication protocol to compute skylines in distributed stream environments. BOCS computes skyline points incrementally in two phases. In the first phase, GridSky computes local skylines on remote sites and only skyline increments on the these sites are sent to the coordinator. In the second phase, the global skyline is obtained by combining remote increments with the latest global skyline.

In [19], Das Sarma *et al.* proposed a set of multi-pass data streaming algorithms that compute the skyline of a massive database with strong worst-case performance guarantees. The data stream in this context refers to the data objects in a database (residing on disk) that is read into and processed through the main memory in a streaming manner. The key contribution of this paper is a randomized multi-pass streaming algorithm called *RAND*. It has two versions: one with fixed windows and another without. Since each pass over the database is time-consuming, RAND minimizes the number of such passes by the use of randomization that helps quickly eliminate a large number of non-skyline points at each pass. The paper shows that single pass algorithms under the sliding window model like [44, 71, 87] are too restrictive and proves that it is impossible to design an efficient skyline algorithm that reads each point exactly once.

Data stream skyline processing under the sliding window model is addressed in [44] and [71]. In this environment, the skyline tends to keep changing with objects arriving and expiring while time passes. An important issue that needs to be addressed here is the expiration of skyline objects, i.e. how to replace expired skyline objects with their proper successor(s) without having to compute from scratch among objects that are exclusively dominated by the expired ones. To handle this problem, Tao *et al.* proposed the *Eager* algorithm [71] that employs an event list, while Lin

et al. developed a method (*StabSky*) that leverages *dominance graphs* [44]. Both these methods memorize the relationship between a current skyline object and its successor(s). Once skyline objects expire, their successor(s) can be presented as the updated skyline without any added computation.

Zhang *et al.* proposed another technique under the sliding window model [87]. The authors developed an incremental method to address continuous, probabilistic skyline queries over sliding windows on uncertain data objects, which have probability thresholds. In addition, the authors extended their techniques to support continuous queries with multiple probability thresholds and probabilistic top-k skyline queries.

In [31], Jiang *et al.* addressed a novel type of query analysis on time series data called *interval* skyline queries. These queries return a set of time series that is not dominated by any other time series in a given time interval. The authors propose two methods to answer these queries: an *On-the-fly* method and a *View-materialization* method. Both methods are shown to be efficient in answering interval skyline queries and incremental maintenance of skylines on updates.

Lastly, Park *et al.* proposed a computation framework called *TI-Sky* [56] that evaluates skyline queries over continuous time-interval streams. The time-interval model is more general than the sliding window model; here, unlike the sliding window model, each object in the stream has its own expiration time. The authors developed a two-layered model of time-based dominance called *macro* and *micro* time-dominance. They further proposed algorithms that effectively exploit the time-dominance model to handle various real-time query operations like insertion, deletion, purging and result retrieval.

The above-mentioned approaches focus on skyline analysis in which the skyline attributes belong to a single stream, thus rendering them inapplicable to the problems being addressed in this proposal. Very recently, Catania *et al.* proposed an algorithm

to process so called *relaxed queries* over data streams [14]. The paper studies a specific case of skyline queries called *relaxation skyline* (r-skyline) queries and extends them to window-based join over data streams. Preliminary experimental results reported in [14] show that there is a need to design more efficient algorithms for r-skyline computation. In particular, mirroring the motivation behind the research presented in this dissertation, the authors conclude that there is a need for the development of algorithms that consider skyline query processing hand-in-hand with window-based join processing.

2.4 Skylines over Imprecise Data Sources

Since the basic assumption of completeness is not applicable to imprecise data, several algorithms have been developed specifically for the efficient handling of skyline queries over imprecise datasets. One category of imprecise data is generally referred to as *uncertain* data. Computing skylines on uncertain data is challenging and more complex than computing skylines on certain data. In uncertain datasets, each tuple has multiple instances or each tuple is associated with a probability density function. In many cases, the probability density function is unavailable and needs to be approximated by examining a set of instances of the uncertain tuple [57]. Hence, skyline processing over uncertain data is expensive, and many algorithms have been proposed to address this problem [57, 3, 45].

A second category of imprecise data is often called *incomplete* data. Khalefa *et. al* were among the first to propose algorithms for the efficient computation of skyline queries over incomplete datasets [34]. The downside of their proposed approach is that it induces *non-transitive* and *cyclic* dominance relationships among the tuples in an incomplete dataset. Other techniques address skylines over incomplete data by using the method of elicitation of missing values [21, 46]. Endres *et. al* proposed

an *insertion strategy* for incomplete data that maintains the transitivity property of the dominance relation [21], however there is an overhead that comes with handling incomplete data in this manner. Another way of handling missing information in skyline computation is by using heuristics that provide default answers for deciding the dominance relationship [46].

Chapter 3

EFFICIENT PROCESSING OF SKYLINE-JOIN QUERIES ON STATIC DATA SOURCES

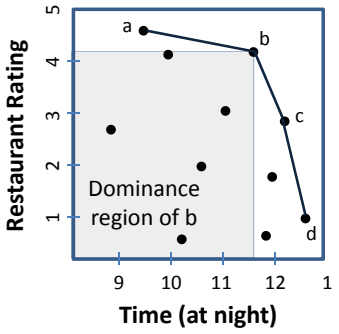
3.1 Introduction

There has been growing interest in efficient processing of skyline queries [13, 70, 77, 11]. The *skyline* of a dataset is the subset of data points that are not dominated[1] by any other data points [13]. Skyline queries are valuable in many multi-criteria decision applications [7] and are becoming a staple of decision support systems. Figure 3.1(a) illustrates an example application, which involves the star-ratings and closing times of a set of restaurants: the points that are connected in the figure represent those restaurants that are part of the skyline set. Other restaurants are not in the skyline because they are dominated in terms of skyline attributes *Time* and/or *Rating* by at least one restaurant that is in the skyline. The shaded area in Figure 3.1(a) is the dominance region of restaurant b: for any restaurant in this range, b is either open till a later time and/or has a better rating.

3.1.1 Skylines over Multiple Static Data Sources

Various algorithms, for example [13, 9, 39], have been developed to address the problem of discovering skylines over a single data source. These early efforts generally assumed that the skyline query is applied to a single table and thus, all the skyline attributes are present in one table. Naturally, these single-source skyline techniques

[1] A point dominates another point if it is as good or better in all dimensions, and better in at least one dimension.



(a) Skyline of late-night restaurants

Restaurants			
Restaurant	*Stars*	*Closing Time*	*Location*
Phoenicia	2	9	Tempe
P.F. Chang's	4	10	Mesa
Thai Rama	3	11	Phoenix
Pho Nhat	5	12	Tempe
Slices	1	11	Phoenix

Bars			
Bar	*Quality*	*Price*	*Location*
Four Peaks	4	6	Tempe
Nook	5	7	Phoenix
The Tavern	3	4	Tempe
The Harp	1	4	Mesa
The Local	2	8	Phoenix

SKYLINE-JOIN OPERATOR
(Join Attribute: Location, Skyline Attributes: Stars, Closing Time, Quality, Price)

Location	*Restaurant*	*Stars*	*Closing Time*	*Bar*	*Quality*	*Price*
Phoenix	Thai Rama	3	11	Nook	5	7
Tempe	Pho Nhat	5	12	Four Peaks	4	6
Tempe	Pho Nhat	5	12	The Tavern	3	4
Skyline-Join Results						

(b) Skyline-join of late-night <restaurant, bar> pairs

Figure 3.1: (a) Example skyline query over a single dataset, (b) Example skyline-join query over two datasets

are not suitable for many applications in which the schema is complex and the skyline attributes are split onto many tables or data sources. This occurs especially in multi-source information integration applications that inherently operate on multiple sources. The task of computing skylines on multiple data sources is commonly known as the *skyline-join* problem.

Example 1. *Figure 3.1(b) shows an example* two-way skyline-join *query. In this example, the attributes used in the skyline query are available from two disjoint data sources and the skyline is defined over the result of join operations performed between the corresponding two tables. The two tables,* Restaurants *and* Bars*, store specific information related to dining and drinking places in different cities. A group of friends planning a fun night consisting of dinner and drinks may be attracted to the information in these tables. They might be interested in finding the best pairs of nearby restaurants and bars, which include restaurants that close late at night and have a high star-rating, and bars that are of high quality and offer low prices.* ◇

A naive approach to processing such queries is to first join the relevant tables

to materialize all candidate tuples, and then, to apply existing single-table skyline algorithms. However, in most cases, a large percentage of the materialized join results will not appear in the final skyline, and thus, a significant portion of the join work done to obtain the combined candidate set will be wasted. If, on the other hand, we could prune the redundant tuples during the join processing (i.e., the join operation is *skyline-sensitive*), then we could achieve significant improvements in the performance of multi-source skyline queries.

Recently, various attempts have been made to tackle this challenge [32, 68, 36, 60, 78]. For instance, the *skyline-join* operator proposed in [68] is a hybrid of the previous skyline and join operators and leverages several pruning opportunities during its operation for faster execution. [78] proposes a non-iterative, single-pass *sort-first-skyline-join (SFSJ)* operation for pruning tuples during the join. Nevertheless, like many others, this operator suffers from several drawbacks, including expensive tuple-to-tuple dominance checks.

3.1.2 Main Contributions

Motivated by aforementioned shortcomings, we propose two non-iterative, single-pass skyline-join algorithms, namely $\mathtt{S^2J}$ and $\mathtt{S^3J}$, that avoid tuple-to-tuple dominance checks wherever possible. The main contributions of this research work are as follows:

- We develop a *skyline-sensitive join* ($\mathtt{S^2J}$) algorithm that relies on a novel *layer/region pruning* (*LR-pruning*) strategy to avoid excessive tuple-to-tuple dominance checks. This algorithm processes *skyline-join* queries over two data sources, namely *outer table* and *inner table*. The key features of the $\mathtt{S^2J}$ algorithm are as follows:
  - The tuples in the outer table are sorted into *layers* of dominance.

  - The inner table tuples are clustered into *regions* based on the Z-values of the skyline attributes.
  - A *trie*-based data structure on the inner table keeps track of the so-called *dominated*, *not-dominated*, and *partially-dominated* regions of the inner table *relative to the layers of the outer table*.
  - $\texttt{S}^2\texttt{J}$ obtains the skyline set by scanning the outer table, only once, while pruning the inner table.

- Next, we propose a *symmetric skyline-sensitive join* ($\texttt{S}^3\texttt{J}$) algorithm that repeatedly swaps the roles of the outer and inner data tables, and which rarely needs to scan any of the input data tables entirely in order to obtain the set of skyline points.

Also, the research presented in this chapter further extends $\texttt{S}^2\texttt{J}$'s and $\texttt{S}^3\texttt{J}$'s two-way *skyline-join* ability to efficiently process *skyline-join* queries over more than two data sources. The extensions are as follows:

- We develop a multi-way (M-way) version of the $\texttt{S}^2\texttt{J}$ algorithm, called $\texttt{S}^2\texttt{J-M}$, that relies on the extended concept of the *LR-pruning* strategy, called M-way *LR-pruning*, to efficiently process *skyline-join* queries over more than two data sources. $\texttt{S}^2\texttt{J-M}$ processes skyline-joins over M data sources that are split into two parts, namely *outer set* and *inner table*. The outer set contains $M-1$ data sources, whereas the data source that is not a part of the outer set is assigned as the inner table. $\texttt{S}^2\texttt{J-M}$'s key features are as follows:
  - The $M-1$ data sources in the outer set are combined into *layers* of dominance using a top-k join operator.

  - $\texttt{S}^2\texttt{J-M}$ obtains skyline-joins by scanning the data sources in the outer set, only once, while pruning the inner table.

- Furthermore, we propose a multi-way (M-way) version of the $\texttt{S}^3\texttt{J}$ algorithm, called $\texttt{S}^3\texttt{J-M}$, that repeatedly swaps the outer set and inner table.

Experimental results show that the proposed algorithms are very efficient and outperform existing skyline-join algorithms on most datasets with different distributions, join rates, dimensions, and cardinalities.

The rest of the chapter is structured as follows: Section 3.2 presents the preliminaries and states the problem tackled in this work. In Sections 3.3 and 3.4, we discuss the proposed *skyline-sensitive join* algorithms in detail. Section 3.5 presents an extensive experimental evaluation of the proposed approaches.

3.2 Problem Definition

Let $p.a_h$ be the value of attribute a_h of tuple p; *p dominates q in the skyline attribute set A_S* ($p \succ_{A_S} q$) when

$$\forall a_i \in A_S, (p.a_i \geq q.a_i) \wedge (\exists a_k \in A_S \mid p.a_k > q.a_k).$$

Intuitively, p is better than or equal to ($\geq$) q in all dimensions of the skyline attribute set A_S and better than ($>$) q in at least one dimension a_k. In the rest of the chapter, we omit the reference to the skyline attribute set and use $p \succ q$ to denote that p *dominates* q (in the corresponding skyline attribute set). We use $p \not\succ q$ for p *does not dominate* q.

Given (a) M datasets, $D_1(a_{1,1}, \ldots, a_{1,d_1}), D_2(a_{2,1}, \ldots, a_{2,d_2}), \ldots, D_M(a_{M,1}, \ldots, a_{M,d_M})$, (b) a set of skyline attributes, $A_S \subseteq \{a_{1,1}, \ldots, a_{1,d_1}\} \cup \{a_{2,1}, \ldots, a_{2,d_2}\} \cup \ldots \cup \{a_{M,1}, \ldots, a_{M,d_M}\}$, and (c) a set of join attributes,

$A_J \subseteq \{a_{1,1}, \ldots, a_{1,d_1}\} \cup \{a_{2,1}, \ldots, a_{2,d_2}\} \cup \ldots \cup \{a_{M,1}, \ldots, a_{M,d_M}\}$, a join-based skyline query seeks to identify a subset of $J = D_1 \bowtie_{A_J} D_2 \bowtie_{A_J} \ldots \bowtie_{A_J} D_M$ consisting of tuples not *dominated* by any other tuples in J. Throughout the chapter, join-based skyline operations are referred to as *skyline-join* operations and we propose novel *skyline-sensitive join (*`SSJ`*)* operators to efficiently support skyline-join query processing[2].

Definition 1. ***Two-way Skyline-Sensitive Join Operator****. A data tuple, p, is in* D_1 $\texttt{SSJ}_{A_J,A_S}$ D_2 *iff (a)* $p \in J = D_1 \bowtie_{A_J} D_2$*, and (b)* $\nexists q \in J - \{p\}$ *s.t.* $(q \succ_{A_S} p)$. $\diamond$

The above definition can be extended to include skyline-join operations over multiple data sources. This definition is given below.

Definition 2. ***M-way Skyline-Sensitive Join Operator****. A data tuple, p, is in* $\texttt{SSJ}(D_1, D_2, \ldots, D_M, A_J, A_S)$ *iff (a)* $p \in Join_{A_J}(D_1, D_2, \ldots, D_M)$*, and (b)* $\nexists q \in J - \{p\}$ *s.t.* $(q \succ_{A_S} p)$. $\diamond$

Example 2. *For example, given two tables* Highly-rated (rName, rating, location, remarks) *and* Business-hours (rName, openingTime, closingTime)*, which contain information about restaurants, the query:*

```
Skyline = SSJ * FROM Highly-rated H,
            Business-hours B,
            WHERE H.rName = B.rName,
            H.rating MAX, B.closingTime MAX
```

[2] As discussed in the related works section, Jin *et al.*, to the best of our knowledge, were the first to define and study the *skyline-join* problem in the context of multi-relational databases [32]; Sun *et al.* extended this work by introducing new pruning criteria [68].

would equi-join the tables on $A_J = \{$Highly-rated.rName, Business-hours.rName$\}$ *and return results that are not dominated by any other results based on the skyline attributes* $A_S = \{$rating, closingTime$\}$. ◇

In this query, the underlying preference function is MAX. While any monotonic function, such as MIN, is also acceptable as the preference function, in the rest of the chapter, without loss of generality, we assume that MAX is specified by the user.

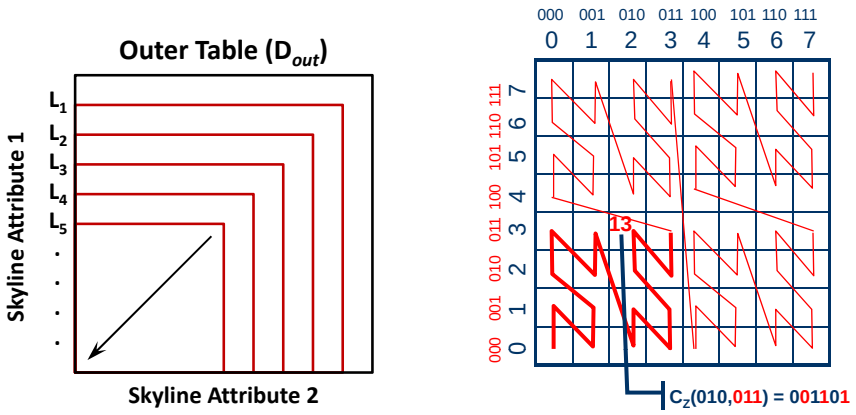
3.3 Processing Skyline-Join Queries over Two Data Sources

Skyline processing on a single table is an expensive operation as it may incur large access costs to perform pairwise dominance checks[3]. If a join operation is required to combine the relevant data, then the query will also incur additional costs for materializing the candidate tuples. If the number of these candidate tuples are large, this will further boost the cost of the skyline processing. Naturally, if the number of tuples materialized and compared can be kept low, significant improvements in the performance of skyline-join queries can be achieved. Thus, we propose a novel *skyline-sensitive join* ($\mathtt{S^2J}$) algorithm in which

- the data in the input tables are ordered in a manner that will help identify skyline points early and prevent unqualified data points from participating in the join operation; moreover,
- whenever possible, the join space needs to be pruned in terms of *blocks* as opposed to *individual* data points, therefore avoiding time-consuming pairwise point-to-point dominance checks.

We achieve these through a novel *layer/region pruning* (*LR-pruning*) strategy that minimizes pairwise tuple comparisons. In LR-pruning, the data in the outer table

[3]A dominance check compares two data points on a dominance condition.



(a) Tuples in *outer* table (D_{out}) are sorted into dominance layers

(b) *Inner* table (D_{in}) tuples are clustered into Z-value regions

Figure 3.2: Layer/region organization for $\mathtt{S^2J}$

is sorted into *dominance layers*, while the inner table is clustered into *regions* using Z-order values of the skyline attributes to support block-based pruning. We also propose a second algorithm, *symmetric skyline-sensitive join (*$\mathtt{S^3J}$*)*, that is similar to $\mathtt{S^2J}$ in principle, but repeatedly swaps the roles of the outer and inner tables. One key outcome of this strategy is that (unlike $\mathtt{S^2J}$, where the outer table is fully scanned), in $\mathtt{S^3J}$, none of the datasets need to be scanned completely in order to obtain the skyline set.

Before we present $\mathtt{S^2J}$ and $\mathtt{S^3J}$, we first provide an overview of the underlying LR-pruning strategy. In the rest of this section, we consider the skyline-join query D_{out} $\mathtt{SSJ}_{A_J,A_S}$ D_{in}, with the MAX preference function.

3.3.1 Layer/Region Organization of Data

Dominance Layering of the Outer Table

Let $A_{S_{out}} \subseteq A_S$ denote the skyline attributes that come from the outer data table, D_{out}. As shown in Figure 3.2(a), the outer table, D_{out}, is sorted into dominance

layers. These dominance layers are similar to the "bands" used in [78] and "skyline layers" in [8].

Definition 3 (Dominance Layer)**.** *Let D_{out} be a data set and A_{Sout} be the corresponding skyline attributes. Each layer, L_i is a subset of D_{out} such that, for any entry in layer, L_i, the maximum of the skyline attribute values is larger than the maximum of the skyline attribute values for any entry in a dominated layer, L_h, i.e.*

$$\forall_{h>i, p\in L_i, q\in L_h} \quad max(p.a|a \in A_{Sout}) > max(q.a|a \in A_{Sout}).$$

◇

Therefore, if two data points in D_{out} are such that $max(p.a|a \in A_{Sout}) = max(q.a|a \in A_{Sout})$, they will be grouped into the same dominance layer. In order to capture the dominance relationship among the layers, we associate to each layer, L_i, a dominance score, $ds(L_i)$:

Definition 4 (Dominance Score)**.** *Let L_i be a dominance layer as defined above. The corresponding dominance score, $ds(L_i)$, is defined as*

$$ds(L_i) = max(p.a|(a \in A_{Sout}) \wedge (p \in L_i)).$$

◇

The dominance layers are obtained by sorting the tuples in the descending order of the value of $ds(L_i)$. The layers are ordered and accessed from the most dominant (with the highest dominance score) to the least dominant (with the smallest dominance score) layer; for example, from L_1 to L_n in Figure 3.2(a).

Region Organization of the Inner Table

Tuples in the inner table, D_{in}, are mapped onto a Z-order curve based on the corresponding skyline attribute set, $A_{Sin} \subseteq A_S$ (Figure 3.2(b)). The Z-order (or Morton-

order [47]) curve, shown in Figure 3.2(b), is a fractal that covers a multi-dimensional data space by repeated applications of the same base pattern, a "Z". The Z-value of a data point can be obtained by interleaving the bits of the binary representation of the coordinate values of the data point. In the example shown in Figure 3.2(b), the Z-value, 001101, of the point $(2, 3)$ is obtained by interleaving the binary representations of 2 and 3, i.e. 010 and 011, respectively.

The Z-curve clusters neighboring points and regions in the space; points nearby in space also tend to be nearby on the curve [86]. For a d-dimensional space, in which the coordinate values fall in the range $[0, 2^v - 1]$, the Z-value of a data point contains $d \times v$ bits grouped into v number of d-bit groups. In the example, we have three 2-bit groups: 00, 11, and 01. Given a Z-value with v many d-bit groups, the first bit group partitions the d-dimensional space into 2^d equi-sized regions (or clusters), the second bit group further partitions each of these 2^d region blocks into 2^d smaller equi-sized sub-regions (or sub-clusters), and so on. For instance, in Figure 3.2(b), all the points located in the lower left quadrant of the space have prefix 00 in their binary representations and span Z-values from 0 to 13. Also, note that, given a prefix of the bit representation of a Z-value, we can determine the minimum ($minZ$) and maximum ($maxZ$) Z-values of that region. $minZ$ is obtained by filling the rest of the bit positions with 0s and $maxZ$ by setting all the missing bits to 1.

As discussed in the related work section, this clustering property of Z-curves has been leveraged for the single-table skyline problem [28, 39]. Note that grids, used for example in [60], and Z-order curves are related in that both partition the data space regularly to cluster nearby points. One key advantage of Z-order based clustering is that the fractal nature of the Z-order curve imposes a pyramid of grids consisting of planes of grids of different sizes on top of each other (grid-plane p has half as many grid cells along each of its dimensions as grid $p + 1$) and, as discussed in the next

Algorithm 1: $\texttt{S}^2\texttt{J}(D_{out},\ D_{in},\ A_S,\ A_J,\ T)$

Input:

D_{out}: Outer table, D_{in}: Inner table, A_S: Set of skyline attributes of D_{out} and D_{in}, A_J: Join attribute set

T: Trie maintained for D_{in}, R: Set of regions to be examined during join-based skyline processing

$ds(L_i)$: Dominance score of current layer in D_{out}

Output:

Skyline result S produced by D_{out} $\texttt{SSJ}_{A_J,A_S}$ D_{in}

Procedure:

Initialize T;

for each layer $L_i \in D_{out}$ scanned from $L_1, L_2, \ldots, L_n$ **do**

if data point scanned is the first data point in L_1 or T has been changed **then**

R = RegionsToExamine(T, $ds(L_i)$);

/* Invoke the `RegionsToExamine` algorithm with parameters T and $ds(L_i)$ only if T has been changed, else R remains the same for the next round. `RegionsToExamine` is also invoked once at the beginning when the first data point in L_1 is scanned */

end if

initialize set S_i, the set of skyline points for L_i;

/* Contains a set of points that do not dominate each other */

for each $r \in R$ **do**

/* r is the Z-value region prefix */

minZval(r) = $minZ(r)$;

maxZval(r) = $maxZ(r)$;

/* Minimum and maximum Z-values of r are obtained */

$joinSet = L_i \bowtie_{A_J} D_{in}$ in the region defined by [minZval(r), maxZval(r)];

add the results $\in joinSet$ to S_i such that only skyline points are obtained in S_i;

for each skyline point $s = s_{out}||s_{in} \in S_i$ **do**

$U = min(s_{out})$;

T = Rmarker(s_{in}, T, U);

/* Update T based on $s_{in} \in D_{in}$ and the bound U obtained by taking the minimum of the coordinate values of $s_{out} \in D_{out}$ */

end for

end for

add S_i to S such that only skyline points are obtained in S;

end for

return S

Figure 3.3: The $\texttt{S}^2\texttt{J}$ Algorithm

section, this property helps us dynamically adapt the sizes of the pruned regions on demand. More specifically, we argue that, *when used along with the dominance layering of the outer table*, a Z-value based organization of the *inner table* can be highly effective in eliminating redundant work for skyline joins. Next, we present the first of our algorithms (namely $\texttt{S}^2\texttt{J}$) based on this observation.

3.3.2 The Skyline-Sensitive Join ($\texttt{S}^2\texttt{J}$) Algorithm

Given two datasets, D_{out} and D_{in}, a set of join attributes, A_J, and a set of skyline attributes, A_S, the $\texttt{S}^2\texttt{J}$ algorithm proceeds as follows:

1. $S_{all} = \emptyset$
2. $\texttt{S}^2\texttt{J}$ scans outer table, D_{out}, from layer L_1 to L_n.
3. For each layer L_i:
   (a) $\texttt{S}^2\texttt{J}$ invokes the `RegionsToExamine(T,` $ds(L_i)$`)` function (see Section 3.3.2) to obtain the corresponding set of Z-value regions, R_i, from the trie structure, T, maintained for the inner table, D_{in}. The Z-value regions of the inner table obtained in this step will participate in the join-based skyline process with L_i.
   (b) $C = \emptyset$
   (c) For each region $r \in R_i$:
      - $C = C \cup (L_i \bowtie_{A_J} r)$ is carried out to combine the outer table tuples in L_i with the inner table tuples in r.
   (d) The skyline set, S_i, of $C \cup S_{all}$ is obtained.
   (e) `Rmarker(`S_i`)` is invoked to mark the appropriate regions of the inner table, D_{in}, based on S_i (see Section 3.3.2).

(f) $S_{all} = S_{all} \cup S_i$

4. $\texttt{S}^2\texttt{J}$ proceeds until all the layers in D_{out} are processed or the entire dataset D_{in} is pruned.

A detailed pseudocode of the $\texttt{S}^2\texttt{J}$ algorithm is presented in Figure 3.3. Intuitively, the $\texttt{S}^2\texttt{J}$ algorithm proceeds from one layer to another in the outer table, and for each outer table tuple considered identifies matching tuples in the inner table. Moreover, as it discovers new skyline tuples among the join results, it prunes regions of the inner table that are no more promising in terms of contributing to the generation of new skyline tuples.

The $\texttt{S}^2\texttt{J}$ algorithm, similar to the SFSJ techniques [78], can report skyline-join results progressively. Any tuple in a layer L_i that has a $ds(L_i)$ value lesser than the $ds(L_j)$ value of a higher dominance layer L_j cannot produce join tuples that dominate any of the existing tuples produced by layer L_j since the layer ordering is monotonic in nature.

In Sections 3.3.2 through 3.3.2, we describe the details of the algorithm. More specifically, we discuss (a) how a Z-curve based trie structure is used for efficiently pruning the inner table, (b) how for each dominance layer of the outer table, the corresponding non-pruned regions of the inner table are identified using the trie, and (c) how a B-tree with a composite key is used to quickly join the outer table tuples in a given dominance layer with the corresponding non-pruned tuples in the inner table. The correctness of the $\texttt{S}^2\texttt{J}$ algorithm is established in Section 3.3.2.

Region-based Pruning of the Inner Table

While it generates new join-tuples and discovers new skyline results among them, in order to *prune regions* on the inner table and to maintain this pruning information in

a scalable manner, S^{2}J calls the Rmarker book-keeping function that marks regions of the inner table (Figure 3.4).

The Rmarker algorithm uses a *trie* data structure to efficiently store marked regions based on their common Z-value region prefixes. Each node of the trie corresponds to a subregion of the space and the sequence of symbols from root to a given node describes the prefix of the Z-order value shared by all points in this region. Consequently, the nodes are used as convenient positions for keeping the dominance markings. More specifically, each node of the trie holds one of the following markings:

- *Not-Dominated* (denoted as ND) – this indicates that a inner table region is not (yet) dominated by any of the skyline points discovered so far.

- *Dominated for U* (denoted as DOM U) – this indicates that a region is dominated; U refers to the *largest* $ds(L_i)$ of the outer table for which the region is guaranteed to be dominated. Intuitively, this marking keeps track of the layers of the outer table for which the Z-value region in the inner table can be considered as pruned.

- *Partially-Dominated* (denoted as PD) – this indicates that a region itself is not dominated, but contains a subregion which is dominated.

The trie is initialized as shown in Figure 3.6(a). As S^{2}J proceeds and new skyline points are found, the trie structure becomes deeper, capturing increasing amount of details. At each invocation of the Rmarker function, the nodes of the trie are considered from the root to the leaves. During this process, if required, the markings of the internal nodes are made *more specific* and/or the leaves of the trie are split to accommodate newly discovered skyline points.

Example 3. *For example, in Figure 3.6(b), the nodes of the initial trie are split and the markings are made more specific in such a way that the root-to-leaf sequence*

Algorithm 2: Rmarker(s_{in}, T, U)

Input:

s_{in}: The tuple in D_{in} for which the trie will be marked, T: Trie structure of D_{in}

U: Bound on the points that join s_{in} to become part of overall skyline, q: Local Queue, l: Maximum depth of T

Output:

Updated Trie T

Procedure:

```
q.enqueue(1); q.enqueue(0);
while q is not empty do
    regionR = q.dequeue();
    if s_in ≻ maxZ(regionR) /* s_in dominates regionR */ then
        if node n in T with prefix regionR is marked as PD or ND or DOM U' such that U > U' then
            mark n in T as DOM U;
        end if
    else if s_in ⊁ minZ(regionR) /* s_in does not dominate regionR */ then
        if n with prefix regionR is not marked PD or DOM U'' and has no ancestor nodes marked as DOM U' then
            mark node n in T as ND;
        end if
    else if s_in ⊁ maxZ(regionR) and s_in ≻ minZ(regionR) /* s_in may dominate few points in regionR */ then
        childRegion0 = prefix regionR appended with 0; childRegion1 = prefix regionR appended with 1;
        if node n in T with prefix regionR is a leaf node and length of prefix regionR in not equal to l then
            /* new child nodes inserted into T, while limiting the depth of T to l */
            mark nodes with prefixes childRegion0 and childRegion1 with the region marker of their parent node n;
            /* Child nodes to be added to T inherit markers of parent node as default markers */
            if node n is marked as ND then
                mark n in T as PD;
                /* Parent node n, originally marked ND, is marked PD since its children are being explored further */
            end if
            insert nodes with prefixes childRegion0 and childRegion1 as children of parent node n in T
            q.enqueue(childRegion0); q.enqueue(childRegion1);
            /* Enqueue the child nodes of parent node n into q to be marked appropriately */
        else if n is an internal node then
            q.enqueue(childRegion0); q.enqueue(childRegion1);
            /* Enqueue the child nodes of parent node n into q to be marked appropriately */
        end if
    end if
end while
return T;
```

Figure 3.4: The `Rmarker` Algorithm

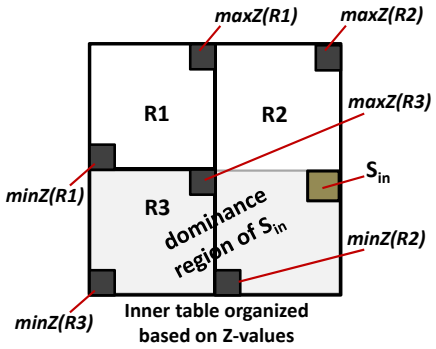
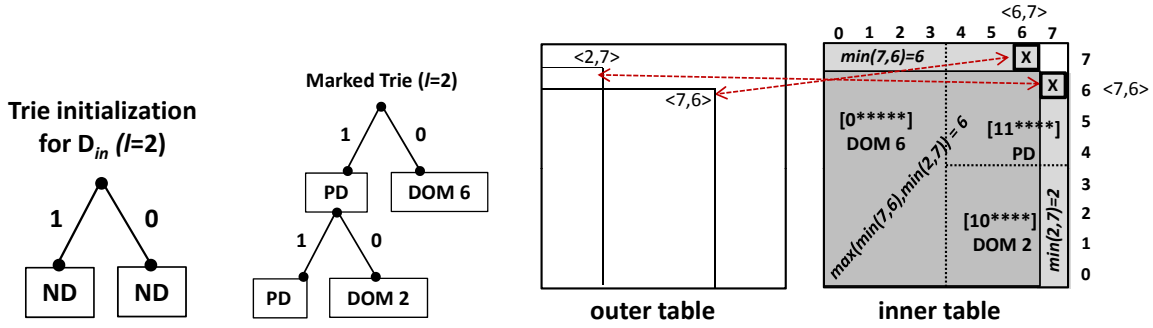


Figure 3.5: Z-value region based dominance test



(a) Initial trie (b) Marked trie (c) Skyline points = $\langle 7, 6, 6, 7\rangle$ and $\langle 2, 7, 7, 6\rangle$

Figure 3.6: (a) Initial trie: it splits the space into two, with the corresponding region markers set to ND (Not-Dominated) as the initial skyline set is empty; (b, c) The trie structure for skyline points $\langle 7, 6, 6, 7\rangle$ and $\langle 2, 7, 7, 6\rangle$; $s_{out} = \{\langle 7, 6\rangle, \langle 2, 7\rangle\}$ and $s_{in} = \{\langle 6, 7\rangle, \langle 7, 6\rangle\}$

"10" represents a subregion of the data space marked "DOM 2" (this marking will be explained in the next subsection). In the same example, the sequence "1" corresponds to another subregion of the data space (which also contains the subregion corresponding to sequence "10") which is marked PD (or "partially dominated"). We describe the process that leads to node split and markings below: ◇

Increasing the Specificity of the Nodes: Let S_i be the skyline result set of layer L_i and let $s = s_{out}||s_{in} \in S_i$ be a skyline point; s_{out} corresponds to a tuple from the outer table, whereas s_{in} corresponds to a tuple from the inner table:

- A node, v, that is originally marked as *Partially-Dominated* (PD) or *Not-*

Algorithm 3: RegionsToExamine(T, $ds(L_i)$)

Input:

T: Trie structure of D_{in}, $ds(L_i)$: Dominance score of current layer in D_{out}

s: Local Stack for DFS, l: Maximum depth of T

Output:

Set of regions R

Procedure:

```
s.push(0); s.push(1);
while s is not empty do
    regionR = s.pop();
    parentMarker = current marker on node n having prefix regionR in T;
    if node n in T is an internal node /* Node n has children */ then
        childRegion0 = prefix regionR appended with 0;
        childRegion1 = prefix regionR appended with 1;
        child0 = marker of node with prefix childRegion0 in T;
        child1 = marker of node with prefix childRegion1 in T;
        if parentMarker is DOM U such that U < ds(L_i) then
            /* Finding a subregion in T marked DOM with the largest U < ds(L_i) */
            if child0 is DOM U_0 such that U_0 > U and U_0 < ds(L_i) then
                /* Explore the branches further */
                s.push(childRegion0); s.push(childRegion1);
            else if child1 is DOM U_1 such that U_1 > U and U_1 < ds(L_i) then
                /* Explore the branches further */
                s.push(childRegion0); s.push(childRegion1);
            else
                add regionR to R;
            end if
        else if parentMarker is PD then
            /* Explore the branches further */
            s.push(childRegion0); s.push(childRegion1);
        end if
    else if parentMarker is ND or PD or DOM U then
        /* node n in T is a leaf node */
        add regionR string to R;
    end if
end while
return R
```

Figure 3.7: The `RegionsToExamine` Algorithm

Dominated (ND) would become dominated if an s_v dominating this region is discovered. Let $S_{in}(v)$ be the set of skyline points on the inner table that dominates this node and $S_{out}(v)$ be the corresponding points on the outer table. Then, the node will be marked as DOM U, where U is

$$U = \max_{s_{out} \in S_{out}(v)} \Big(\min_{a \in A_{S_{out}}} (s_{out}.a) \Big).$$

Intuitively, U is the bound on the $ds(L_i)$ of the layers of the outer table for which the region is guaranteed to be dominated. Consequently, this region need not be considered beyond any layer where $ds(L_i) \leq U$.

- A node that is marked *Dominated* for U (DOM U) can be remarked as DOM U' if a bound $U' > U$ is found. Consequently, the algorithm progressively tightens the bound corresponding to a region, therefore pruning the region for an increasing number of outer table layers.

Node Split: A leaf node may need to be split when a subregion of the node is found to be dominated by a new skyline point. If the leaf was originally marked *Not-Dominated* (ND), then it will now be marked as *Partially-Dominated* (PD) and two child nodes will be added to this node. If the leaf was marked *Dominated for U* (DOM U), it will be split *only if* the subregion has an upper bound U' larger than U. When a leaf is split, the new child nodes are inserted into a queue and are examined and marked appropriately.

In order to control the resolution of the markings, we define a parameter l to constrain the depth of the trie: if a node is at the maximum defined depth l, then that node is not split any further. Intuitively, if we set l to a very small value, markings will cover very large regions and therefore may not help prune the space as it may difficult to completely dominate a large portion of the data space – consequently, this

may lead to large numbers of dominance checks. If the depth of the trie is allowed to become large, on the other hand, then each leaf of the trie data structure will represent an individual data point, which in turn may enable finer pruning decisions at the cost of increased trie management overhead. The effects of the depth of the trie structure (and of the resulting granularity of the regions maintained by the trie) on our skyline-join approach are studied experimentally in Section 3.5.2.

Dominance Checks: Both of the above operations involve region-based dominance checks as in [39]. To implement these dominance checks efficiently, the `Rmarker` algorithm performs Z-value region based dominance tests. Let R be the prefix that represents a Z-value region:

1. If $s_{in} \succ maxZ(R)$, then $s_{in} \succ R$.
2. If $s_{in} \not\succ maxZ(R) \wedge s_{in} \succ minZ(R)$, then some points in region R may be dominated by s_{in}; hence this region needs to be explored further.
3. If $s_{in} \not\succ minZ(R)$, then $s_{in} \not\succ R$.

Figure 3.5 shows the region-based dominance tests of $R1$, $R2$, $R3$ against point s_{in}. Here, based on the above conditions, s_{in} dominates region $R3$ and does not dominate region $R1$. Region $R2$ needs to be explored further, since s_{in} might dominate some of the points in this region.

Example 4. *Figures 3.6(b) and 3.6(c) show the state of the trie ($l = 2$) after* `Rmarker` *is executed for the skyline points $\langle 7, 6, 6, 7\rangle$ and $\langle 2, 7, 7, 6\rangle$, where $\langle 7, 6, 6, 7\rangle$ is obtained by combining $\langle 7, 6\rangle$ from the outer table with $\langle 6, 7\rangle$ from the inner table and $\langle 2, 7, 7, 6\rangle$ is got by joining $\langle 2, 7\rangle$ with $\langle 7, 6\rangle$. Each of the trie nodes are labeled based on their overall coverage. For instance, the node with prefix 10 is labeled DOM 2 because a*

skyline point dominates the Z-value region with prefix 10 and there is a subregion that only $\langle 2, 7, 7, 6\rangle$ *dominates, implying that* $U = min(2, 7) = 2$. ◇

Fetching Non-Pruned Regions from the Trie

As the `S²J` algorithm proceeds from one dominance layer to another in the outer table, for each layer, L_i, the algorithm calls the `RegionsToExamine` function to identify the corresponding non-pruned regions of the inner table (Figure 3.7).

The input to this function is the $ds(L_i)$ value of the current outer table layer L_i. Given this value, `RegionsToExamine` returns the regions corresponding to the nodes of the trie that are currently marked ND and the regions related to the nodes that are marked DOM U with the largest $U < ds(L_i)$. Note that, if a region corresponds to a leaf node of the trie and this node is currently marked PD, then this region is returned as well.

Example 5. *Given a layer* L_i *with* $ds(L_i) = 7$ *from the outer table, the trie in Figure 3.6(b) would return all the Z-value regions represented by the leaf nodes, i.e. Z-value regions with prefixes 11, 10 and 0. In contrast, given a layer* L_i *with* $ds(L_j) = 4$*, the trie structure would only return the nodes that represent Z-value regions with prefixes 11 and 10.* ◇

Note that, if a region r is marked *Dominated for* U and if $U \geq ds(L_i)$, then this region does not participate in the join with the current layer L_i. This is because the layers are considered in the decreasing order of $ds(L_i)$ and the bound U on a region monotonically increases; thus, once eliminated, a region will never be considered for any of the subsequent layers. This ensures that entire regions are pruned and the parts of the inner table that are pruned grow over time.

Layer-to-Region Joins

Once the non-pruned regions from the inner table are identified, S²J joins the data in the current layer of the outer table with the data in these non-pruned regions. To enable the join to be processed in an efficient manner, we construct a B-tree index on the inner table, D_{in}, built using a composite key

$$s_{in}.A_{J_{in}} : Z(s_{in}.A_{S_{in}}),$$

where $s_{in}.A_{J_{in}}$ is the values of the tuple $s_{in} \in D_{in}$ for the join attributes $A_{J_{in}}$ and $Z(s_{in}.A_{S_{in}})$ is the Z-value corresponding to the skyline attribute values of s_{in}. Thus, given a join attribute value and a prefix of the relevant region in the inner table, we can quickly identify matching skyline attribute values from the B-tree index: more specifically, in order to join data tuples from the outer table layer L_i with the inner table region r, we first compute $minZ(r)$ and $maxZ(r)$ values of the region r (see Section 3.3.1) and then perform the join operation

$$\left(L_i \bowtie_{\Theta(A_{J_{out}}, A_{J_{in}}), [minZ(r), maxZ(r)]} D_{in}\right)$$

using the composite search key of the B-tree.

Soundness and Completeness of S²J

The S²J algorithm (Figure 3.3) is correct in that, given two datasets, D_{out} and D_{in}, a set of join attributes, A_J, and a set of skyline attributes, A_S, the S²J algorithm returns a set of results compatible with Definition 5 – i.e. it is sound (does not produce any non-skyline results) and complete (does not miss any skyline results).

Soundness: We first establish the soundness of S²J. Let $s = s_{out}||s_{in}$ be returned as a skyline point. Then, there is no $s' = s'_{out}||s'_{in}$ such that $s'_{out} \succ s_{out}$ and $s'_{in} \succ s_{in}$.

Proof. Let us assume there exists such an s'. Note that s'_{out} will be in a layer, L', dominating or equal to the layer, L, of s_{out}; therefore, $ds(L') \geq ds(L)$.

- Since $s_{out}||s_{in}$ has been returned, when layer L has been considered, s_{in} was either not dominated or was dominated with a bound U where $ds(L) > U$.
- Since $ds(L') \geq ds(L)$, s'_{out} is seen before s_{out}.
- Since when s_{out} was considered, s_{in} was either not dominated or was dominated with a bound U, when s'_{out} was considered earlier, s'_{in} (which dominates s_{in}) must also be either not dominated or dominated with a bound $U' \leq U$ (due to monotonicity of bounds).
- Since for s'_{out} we have $ds(L') \geq ds(L)$, where $ds(L) > U$, and s'_{in} must be either not dominated or dominated with a bound $U' \leq U$, it follows that when s'_{out} is considered, s'_{in} has not yet been pruned.

Therefore, $s' = s'_{out}||s'_{in}$ which dominates $s = s_{out}||s_{in}$ would have been enumerated earlier than s, and thus, s' would have pruned s, contradicting the premise that s has been identified as a skyline point. Hence, there cannot be an s' in the skyline result. This proves that the $\mathtt{S^2J}$ algorithm is sound. □

Completeness: We now establish the completeness of $\mathtt{S^2J}$. Let $t = t_{out}||t_{in}$ be a join result that has not been included in the skyline result. Thus, there must be an $s = s_{out}||s_{in}$, where $s_{out} \succ t_{out}$ and $s_{in} \succ t_{in}$, that is returned.

Proof. Let us assume that there does not exist such an s. Let L be the layer in which t_{out} is considered.

- Since t has not been included in the result, when layer L is considered, t_{in} must be in a region dominated with a bound $U \geq ds(L)$.

- Since t_{in} is in a region dominated with $U \geq ds(L)$, there must have been an $s' = s'_{out}||s'_{in}$ where $s'_{in} \succ t_{in}$ and $ds(L') > ds(L)$ (this is due to the monotonically decreasing property of $ds(L)$ values).

- Since there does not exist $s \succ t$ and since $s'_{in} \succ t_{in}$, then $s'_{out} \not\succ t_{out}$; i.e., $ds(L') \leq ds(L)$.

The last two statements above contradict each other; thus, there must exist an $s \succ t$, and hence, t cannot be in the skyline. This proves the completeness of $\mathtt{S^2J}$. $\square$

Time Complexity of $\mathtt{S^2J}$

The execution cost of the $\mathtt{S^2J}$ operator consists of two major components: (1) the time, $cost_{prep}$, needed to construct the dominance layers of the outer table and relevant data structures on the inner table and (2) the time, $cost_{comp}$, needed to compute the skyline join results.

Preparation Costs The first of the above-mentioned costs, $cost_{prep}$, includes the sorting time needed to create the dominance layers on the outer data table and the time needed to create the B-tree used for producing skyline candidates. Consequently, $cost_{prep}$ can be computed as

$$cost_{prep} = \underbrace{O(n \times log\ n)}_{sorting} + \underbrace{O\Big(m \times log\ m + \sum_{h=0}^{log_f\ m} \frac{m}{f^h}\Big)}_{indexing},$$

where n is is the number of tuples in the outer relation, m is the number of tuples in the inner relation, and f is the fanout of the tree – assuming that the B-tree is created using bulk-loading B-tree strategy, which first sorts all the data and than constructs the tree one ayer at a time from bottom up.

Note that the sorting step is performed on the skyline attributes of the outer data set. Therefore, a given sorted order can be re-used for all queries, where the same skyline attribute set is used for the outer data set. Similarly, the B-tree with the composite search key is created on the combination of the join attributes and the (Z-values of the) skyline attributes. Therefore, the B-tree can be re-used for different queries with the same join and skyline attributes for the inner-table.

Candidate Enumeration and Evaluation Costs The time, $cost_{comp}$, needed to compute the skyline join results on the other hand includes the following: The $\texttt{S}^2\texttt{J}$ operator scans the outer table once, and for each layer, L_i, it (a) identifies the relevant regions (that are either not dominated yet or dominated with a bound $U < ds(L_i)$), (b) performs a join operation with these relevant regions of the inner table using a B-tree, and (c) finds the skyline based on the resulting candidates. Therefore, $cost_{comp}$ is a function of the number of layers in the outer table, the size of the trie used for discovering the relevant regions, and the amount of pruning achieved based on the data distribution:

$$cost_{scan}(n) + \sum_{i=1}^{\#\ outer\ layers} cost_{INLJ}\Big(ls(L_i), nip\big(l, ds(L_i)\big)\Big),$$

where

- n is the number of tuples in the outer relation,
- $\#\ outer\ layers$ is the number of layers in the outer relations,
- $ls(L_i)$ is the number of tuples in the outer layer, L_i,
- $ds(L_i)$ is the dominance score corresponding to outer layer, L_i,
- $nip(l, B)$, is the number of inner pages that are either not dominated yet or dominated with a bound B given the trie with depth l,

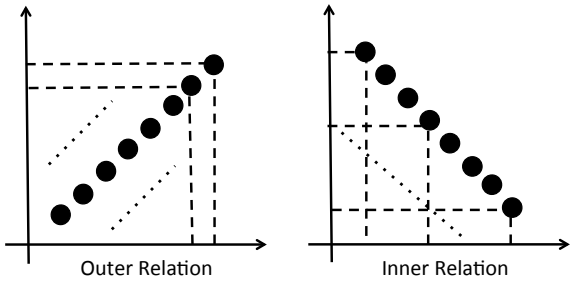


Figure 3.8: A sample worst-case scenario where S^{2}J cannot benefit from layering in the outer relation or region-prunning in the inner relation

- $cost_{scan}(x)$ is the cost of scanning x tuples, and
- $cost_{INLJ}(x, p)$ is the cost of performing index nested loop join of x tuples in the outer relation, with p unpruned pages in the inner table.

In Section 3.5, we study the effect of various parameters (including the maximum allowed trie depth, l) on the performance of the S^{2}J algorithm.

Note that, in the extreme case, each data point in the outer table belongs to a distinct layer and the inner-table is anti-correlated – consequently, the algorithm cannot benefit from region pruning of the inner-table (Figure 3.8). The complexity in the extreme case would be the cost of scanning the leaves stored in the trie structure of the inner table in order to discover the non-pruned tuples for each of the n tuples in the outer table, leading to an $O(n \times m)$ worst-case data access cost, assuming (again the worst case situation) that each of the m tuples in the input table is a lead in the trie structure.

Space Complexity of S^{2}J

In addition to storing the candidate tuples generated during the process, which in the worst case can be $O(n \times m)$ where n is the number of tuples in the outer table and m is the number of tuples in the inner table, the S^{2}J algorithm has to maintain two distinct data structures for the inner table: a B-tree that supports efficient candidate

enumeration and a trie that maintains the dominance markings. This leads to the following worst-case complexity:

$$cost_{space} = \underbrace{O\Big(n \times m\Big)}_{Candidates} + \underbrace{O\Big(\sum_{h=0}^{log_f\ m} \frac{m}{f^h}\Big)}_{Btree} + \underbrace{O(min\{2^t, m\})}_{trie},$$

where m is the number of tuples in the inner table, f is the fanout of the Btree and, t is the bound on the height of the trie.

Note that since the outer table is processed one layer at a time, in practice, the number of candidates that need to be maintained at a given point time is governed with the size of the current outer layer and the number of matching tuples in the inner table. The group skyline optimization discussed in Section 3.4.4 further reduces the number of candidates that need to be enumerated.

3.3.3 The Symmetric Skyline-Sensitive Join (S³J) Algorithm: Table Swapping and Early Stopping

The S²J algorithm described above helps reduce the amount of work by supporting region-pruning on the inner table. It, however, has to scan the outer table in its entirety. The S³J algorithm (Figure 3.9) is similar to the S²J, but further reduces the skyline-join cost by pruning some of the dominance-layers from consideration. To achieve this, given two datasets, D_1 and D_2, the S³J algorithm maintains dominance layers and trie structures for both D_1 and D_2, and swaps the roles of the outer and inner tables for each dominance layer of D_1 and D_2

This swapping enables both tables to be progressively pruned relative to each others' layers, and supports a stopping condition not available to S²J (Figure 3.10). Let D_{out} be the current table that serves as the outer table and D_{in} be the current inner table. Let L_i^{out} denote the current layer being considered for D_{out} and L_h^{in} denote the (most recently considered) layer for D_{in}. For layer L_i^{out}, S³J considers the

Algorithm 4: $\mathtt{S^3J}(D_{out}, D_{in}, A_S, A_J)$

Input:

D_{out}: Outer table, D_{in}: Inner table, A_S: Set of skyline attributes of D_{out} and D_{in}

A_J: Join attribute set, T_{in}: Trie maintained for D_{in}, T_{out}: Trie maintained for D_{out}

R: Set of regions to be examined during join-based skyline processing

$ds(L_i)$: Dominance score of current layer in D_{out}

Output:

Skyline result S produced by D_{out} $\mathtt{SSJ}_{A_J,A_S}$ D_{in}

Procedure:

Initialize T_{in}, T_{out};

for each element e in layer $L_i^{out} \in D_{out}$ and $L_i^{in} \in D_{in}$ **do**

$p(L_i^{out}) = \langle ds(L_i^{out}), ds(L_i^{out}), \ldots, ds(L_i^{out}) \rangle$

find the region marker m_{out} for $p(L_i^{out})$ in T_{out}

if m_{out} is ND or

the largest bound U_{max} for m_{out} returned as DOM such that $U_{max} < ds(L_i^{in})$ **then**

$S_i^{out} = \mathtt{S^2J}(e(L_i^{out}), D_{in}, A_S, A_J, T_{in})$;

/* S_i^{out} is skyline result set for $L_i^{out} \in D_{out}$ returned by $\mathtt{S^2J}$ */

else

stop scanning D_{out}

end if

$p(L_i^{in}) = \langle ds(L_i^{in}), ds(L_i^{in}), \ldots, ds(L_i^{in}) \rangle$

find the region marker m_{in} for $p(L_i^{in})$ in T_{in}

if m_{in} is ND or

the largest bound U_{max} for m_{in} returned as DOM such that $U_{max} < ds(L_i^{out})$ **then**

$S_i^{in} = \mathtt{S^2J}(e(L_i^{in}), D_{out}, A_S, A_J, T_{out})$

/* S_i^{in} is skyline result set for $L_i^{in} \in D_{in}$ returned by $\mathtt{S^2J}$ */

else

stop scanning D_{in}

end if

add S_i^{out} and S_i^{in} to S such that only skyline points are obtained in S;

end for

return S

Figure 3.9: $\mathtt{S^3J}$ swaps the outer and inner tables in a round-robin manner

extremum (or corner) point $p(L_i^{out}) = \langle ds(L_i^{out}), ds(L_i^{out}), \ldots, ds(L_i^{out}) \rangle$ of the layer and checks (using the trie structure corresponding to table D_{out}) if this extremum point is being dominated for any of the layers of D_{in} considered so far:

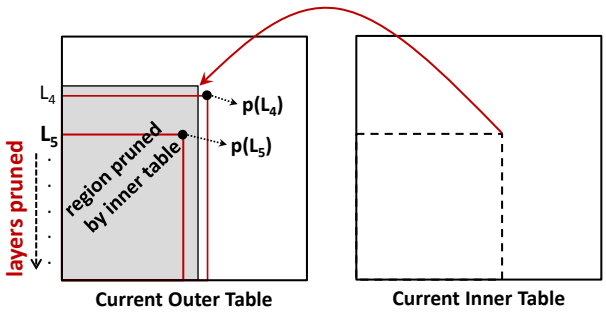


Figure 3.10: Scanning the (current) outer table stops at layer L_5 because L_5 and all the following layers are pruned by the (current) inner table

- If $p(L_i^{out})$ is not dominated, then the layer, L_i^{out}, is considered as before.
- If $p(L_i^{out})$ is dominated, then S^{3}J finds the largest bound U_{max} for which $p(L_i^{out})$ is dominated in the trie and checks if $U_{max} \geq ds(L_h^{in})$, where L_h^{in} is the current layer for the inner table, D_{in}. If so, then the data layer, L_i^{out}, and all the subsequent layers of the outer table can be eliminated from further consideration, and hence, the consideration of this table as an outer table and its scanning can be stopped.

The correctness of the early stopping condition is established next.

Correctness of S^{3}J

Here, we establish the correcteness of S^{3}J, relying on the correctness of S^{2}J established in the previous section.

Soundness: S^{3}J relies on the S^{2}J algorithm described in Section 3.3.2 to generate the skyline tuples and the soundness of S^{3}J comes trivially from the soundness of S^{2}J.

Completeness: We now establish that the early stopping condition does not cause a violation of the completeness of S^{3}J. Let $t = t_{out}||t_{in}$ be a join result that has not been included in the skyline result. Since S^{2}J is complete, we know that if the layer containing t_{out} has been seen, then there must exist an $s \succ t$ found earlier (see

Section 3.3.2). We need to show that if t_{out} is not found due to early stopping, then there must also exist $s \succ t$.

Proof. Let L_{out} be the layer in which t_{out} would be considered if the early stop condition was not applied.

- Since the early stopping condition has been applied and t_{out} has been pruned, we know that $p(L_{out})$ is dominated in the corresponding trie.
- Let U^{in} be the largest bound on the dominated regions covering $p(L_{out})$. Since the early stop condition has been applied and t_{out} has been pruned, we also know that $U^{in} > ds(L_{in})$, where L_{in} is the current layer for the inner table.
- Since L_{in} is the current layer for the inner table, it includes t_{in}.
- Since

  $$U^{in} = \max_{s_{in} \in S_{in}} \left(\min_{a \in A_{S_{in}}} (s_{in}.a) \right)$$

  and since

  $$U^{in} \geq ds(L_{in})$$

  it follows that there exits at least one $s_{in} \in S_{in}$ that dominates $p(L_{in})$, which also implies that $s_{in} \succ t_{in}$.
- Since s_{in} is used in the computation of U^{in} on the region dominating t_{out}, there must be a corresponding $s_{out} \succ t_{out}$.
- Therefore, there exists an $s = s_{out}||s_{in}$, where $s_{out} \succ t_{out}$ and $s_{in} \succ t_{in}$.

In other words, there exits $s \succ t$ for any t that is eliminated from consideration due to the early stopping condition. This proves that $\mathtt{S^3J}$ is complete. □

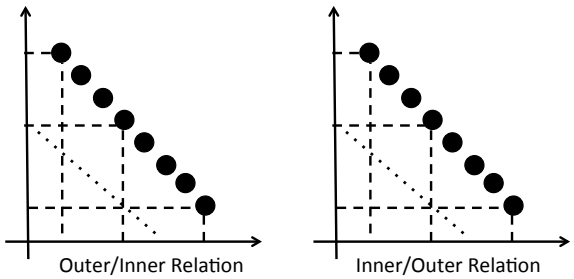


Figure 3.11: A sample worst-case scenario where S³J cannot benefit from region-prunning for either relation

Time Complexity of S³J

The preparation cost $cost_{prep}$, for the S³J algorithm is similar to the cost of the S³J algorithm, except that the work has to be done twice for each table:

$$cost_{prep} = O\Bigg(\sum_{i=\in\{1,2\}} \underbrace{(n_i \times log\ n_i)}_{sorting} + \underbrace{\Big(n_i \times log\ n_i + \sum_{h=0}^{log_f\ n_i} \frac{n_i}{f^h}\Big)}_{indexing} \Bigg),$$

where n_i is the size of the i^{th} table.

To assess the time, $cost_{comp}$, needed to compute the skyline join results, let us first consider the sample worst-case scenario for the S²J algorithm shown in Figure 3.8. It is easy to see that swapping tables for each layer will help significantly in this case, because when the tables are swapped, the inner relation is no more anti-correlated and thus there are larger opportunities for LR-pruning. In contrast, the worst case for S³J occurs when both tables are anti-correlated (Figure 3.11): Consider a scenario where we are given two tables that each contain data which are themselves skylines. On joining these tables (on a non-skyline attribute), each tuple in the join result will also be a global skyline point. This scenario constitutes the worst-case for S³J: since both the input data as well as the output consists of all skylines, there is no pruning or early stop possible. This implies that a full join of the input tables has to be performed (with the help of the index structure). It is easy to that in this scenario, both tables will need to be scanned and joined with the other to produce skyline

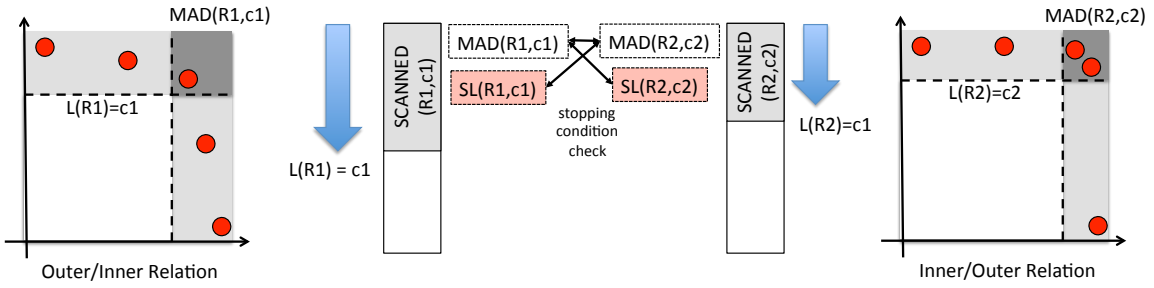


Figure 3.12: Overview of the SFSJ algorithm [78] (the red dots indicate the current skyline objects local to each table)

results. this implies that, while in general $\mathtt{S^3J}$ avoids the pitfalls of $\mathtt{S^2J}$, the worst-case cost of $\mathtt{S^3J}$ (when the skyline attributes for both relations are anti-correlated) can be worse than that of $\mathtt{S^2J}$.

[78] introduced the concept of *instance optimality* for skyline-join operation, in terms of how quickly the algorithms stop scanning the input tables. [78] has also shown that (a) the SFSJ algorithms are instance optimal with a ratio of 2 for cases where the inputs have at most K tuples in each "band" and (b) the algorithms are not instance optimal in the general case. Here, we first review the concept of instance optimality as defined by [78] and then show that for any pair of data tables, the $\mathtt{S^3J}$ algorithm stops as early as SFSJ.

Instance Optimality: [78] defines instance optimality as follows: Let $\mathcal{A}$ be a class of skyline join algorithms and let $totDepth(X, I)$ be total depth to which the algorithm $X \in \mathcal{A}$ scans the two input tables in data instance I. An algorithm $A \in \mathcal{A}$ is *instance optimal* if there exist constants c and c' such that $totDepth(A, I) \leq c \times totDepth(B, I) + c'$ for any instance data instance I and algorithm $B \in \mathcal{A}$.

$\mathtt{S^3J}$ Stops as Early as SFSJ: To see why $\mathtt{S^3J}$ stops as early as SFSJ, let us first view how SFSJ operates (Figure 3.12): As in $\mathtt{S^3J}$, SFSJ also scans the two input relations layer-by-layer and alternates between the two tables at each iteration. Unlike $\mathtt{S^3J}$ (visualized in Figure 3.13), however, SFSJ does not prune the relations. Instead,

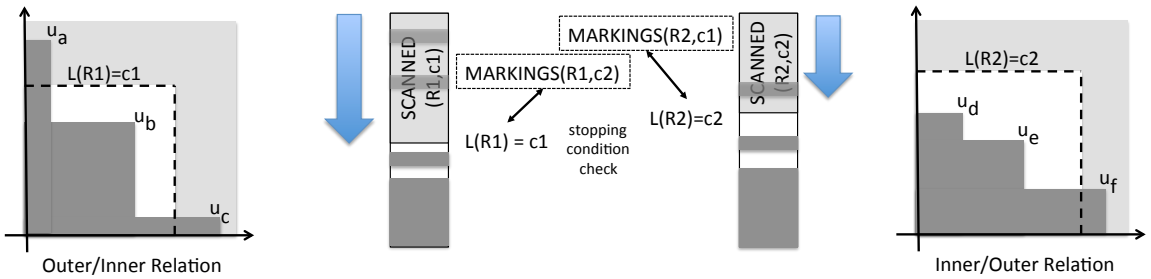


Figure 3.13: Overview of the **S^{3}J** algorithm (the dark grey regions indicate part of the tables that are pruned according to the current markings on the relations – note that each marking comes with a bound for which the region is dominated)

SFSJ operates as follows: Let the current layer for relation R_1 be $L(R_1) = c_1$ and the current layer for relation R_2 be $L(R_2) = c_2$. Then, for each relation, R_i, that the SFSJ method maintains

- a *maximum anti-dominated* region $MAD(R_i, c_i)$, consisting of points that have larger values than the corner of the current layer $L(R_i)$ in all dimensions (i.e., the upper-right regions in Figure 3.12); and
- a *current local skyline* region $SL(R_i, c_i)$, consisting of local skyline points among the points so far seen in region R_i (i.e., the red points in Figure 3.12).

The global skyline candidates are generated by joining the current local skyline objects, $SL(R_i, c_i)$, in one relation with the tuples seen so far in the other relation ($SCANNED(R_j, c_j)$ – i.e., the light and dark grey regions in Figure 3.12). In addition, [78] also shows that scanning of the relations can stop when the following stopping condition is satisfied:

$$(\pi_\alpha(SL(R_1, c_1)) \quad - \quad \pi_\alpha(MAD(R_2, c_2)) = \emptyset) \quad \wedge$$
$$(\pi_\alpha(SL(R_2, c_2)) \quad - \quad \pi_\alpha(MAD(R_1, c_1)) = \emptyset),$$

where α are the join attributes and "$-$" is the set difference operation. In particular, authors show that the remaining tuples in relation $R_i \in \{R_1, R_2\}$ will not

contribute to skyline generation if, for $R_j \in \{R_1, R_2\}$ s.t. $R_j \neq R_i$, the following holds: $MAD(R_j, c_j) \neq 0$ and $\pi_\alpha(SL(R_j, c_j)) - \pi_\alpha(MAD(R_i, c_i)) = \emptyset$. In this work, we establish that S^{3}J stops as early as SFSJ by showing that for any data instance $I = \{R_1, R_2\}$ and pair of layers $L(R_1) = c_1$ and $L(R_1) = c_2$ for which the SFSJ stopping condition is satisfied, the S^{3}J stopping condition would also be satisfied.

S^{3}J *stops as early as SFSJ.* We will prove this using contradiction. Let $I = \{R_1, R_2\}$ be a data instance and layers $L(R_1) = c_1$ and $L(R_1) = c_2$ be such that the following holds: $MAD(R_2, c_2) \neq \emptyset$ and $\pi_\alpha(SL(R_2, c_2)) - \pi_\alpha(MAD(R_1, c_1))$; i.e., SFSJ stops scanning of the relation R_1.

On the other hand, let us also assume that S^{3}J is not able to stop the scanning of relation R_1 at this situation. In other words, either

- *Case 1:* the extremum, $p(L(R_1))$, of the current layer, $L(R_1)$, is not in a region marked dominated, or
- *Case 2:* $p(L(R_1))$ is in a region marked dominated, but the largest bound U_{max} for which $p(L(R_1))$ is marked dominated is such that $U_{max} < ds(L(R_2))$.

We will show that each of these cases leads to contradiction. Let us first consider the first case:

Case 1: If the extremum, $p(L(R_1))$, is not marked dominated, then there cannot be a tuple $t_1 \in MAD(R_1, c_1)$ which joins with some other tuple in $t_2 \in SCANNED(R_2, c_2)$ to produce a skyline. This, however, conflicts with the fact that $\pi_\alpha(SL(R_2, c_2)) - \pi_\alpha(MAD(R_1, c_1))$ as this would imply either that

- $SL(R_2, c_2) = \emptyset$ or that
- there is a $t_2 \in SL(R_2, c_2) \subseteq SCANNED(R_2, c_2)$ which would join with a tuple $t_1 \in MAD(R_1, c_1)$.

But, since $(MAD(R_2, c_2) \neq \emptyset) \rightarrow (SL(R_2, c_2) \neq \emptyset)$, if $\pi_\alpha(SL(R_2, c_2)) - \pi_\alpha(MAD(R_1, c_1))$, then it *must be the case that* there exists $t_2 \in SL(R_2, c_2) \subseteq SCANNED(R_2, c_2)$ which would join with a tuple $t_1 \in MAD(R_1, c_1)$. Thus, this leads to a contradiction with the fact that $SFSJ$ reached the stopping condition.

Case 2: If $p(L(R_1))$ is in a region marked dominated, but the largest bound U_{max} for which $p(L(R_1))$ is marked dominated is such that $U_{max} < ds(L(R_2))$, then it must be the case that there exists a tuple $t_1 \in MAD(R_1, c_1)$ which joins with some other tuple in $t_2 \in SCANNED(R_2, c_2)$ to produce a skyline. Let $\mathcal{T}$ be the set of such $\langle t_1, t_2 \rangle$ pairs. Then, we have

$$U_{max} = \max_{\langle t_1, t_2 \rangle \in \mathcal{T}} \left(\min_{a \in A_{S_2}} (t2.a) \right) < ds(L(R_2)),$$

which implies that none of the t_2 tuples is in the region $MAD(R_2, c_2)$. But, since

- $(MAD(R_2, c_2) \neq \emptyset) \rightarrow (\exists t \in (SL(R_2, c_2) \cup MAD(R_2, c_2))$ and
- $(\exists t_2 \in (SL(R_2, c_2) \cup MAD(R_2, c_2)) \wedge (\pi_\alpha(SL(R_2, c_2)) - \pi_\alpha(MAD(R_1, c_1))) \rightarrow (\exists \langle t_1, t_2 \rangle \ (t_1 \in MAD(R_1, c_1)) \wedge (t_2 \in MAD(R_2, c_2)) \wedge (\pi_\alpha(t_2) = \pi_\alpha(t_1)))$.

the fact that SFSJ reached the stopping condition for R_1 implies that there *must exists* a $\langle t_1, t_2 \rangle \in \mathcal{T}$ such that t_2 is in the region $MAD(R_2, c_2)$. Thus, Case 2 also leads to a contradiction with the fact that the $SFSJ$ method reached the stopping condition for R_1.

Conclusion: Since both of the cases that would need to be satisfied for $\mathtt{S^3J}$ keep scanning relation R_1 when SFSJ stops scanning it leads to contradictions, we can conclude that whenever SFSJ stops scanning relation R_1, $\mathtt{S^3J}$ also stops scanning R_1. Note that similar proof also holds for the stopping condition for R_2 and these together completes the proof of the statement "$\mathtt{S^3J}$ stops as early as SFSJ". □

The property of S^{3}J to stop as early as SFSJ implies that S^{3}J instance optimal in all situations where SFSJ is instance optimal:

Corollary 1 (Instance Optimality of S^{3}J)**.** S^{3}J *instance optimal in all situations where SFSJ is instance optimal.* ◇

The fact that S^{3}J stops as early as SFSJ is also observed in the experiments presented in Section 3.5: S^{3}J scans less than (and performs as good or better than) SFSJ. Since S^{3}J is able to make pruning decisions more proactively than SFSJ, S^{3}J needs to see less data before it can stop scanning the input relations.

Space Complexity of S^{3}J

Like the S^{2}J algorithm, S^{3}J has to maintain two distinct data structures, but for both of the tables: a B-tree that supports efficient candidate enumeration and a trie that maintains the dominance markings. This leads to the following worst-case complexity:

$$cost_{space} = \underbrace{O\Big(n_1 \times n_2\Big)}_{Candidates} + O\Bigg(\sum_{i \in \{1,2\}} \underbrace{\Big(\sum_{h=0}^{log_f \ n_i} \frac{n_i}{f^h} \Big)}_{Btree} + \underbrace{(min\{2^t, n_i\})}_{trie} \Bigg),$$

where n_i is the number of tuples in table i and t is the bound on the height of the trie.

As noted earlier in Section 3.3.2, since the outer table is processed one layer at a time, in practice, the number candidates that need to be maintained at a given point time is much smaller.

3.4 Processing Skyline-Join Queries over More than Two Data Sources

The *skyline-sensitive join (*SSJ*)* algorithms, namely S^{2}J and S^{3}J, presented in the previous section focus on processing skyline-join queries over pairs of data sources. In

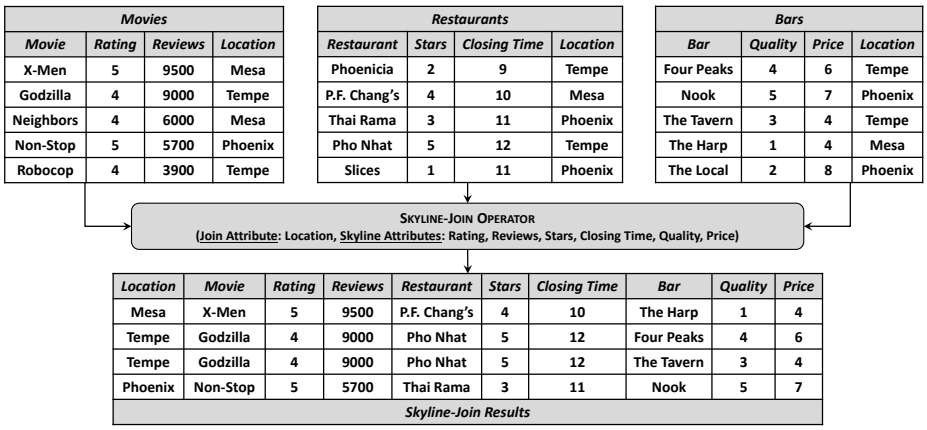


Figure 3.14: An example multi-way skyline-join query

this section, we extend $\mathtt{S^2J}$'s and $\mathtt{S^3J}$'s two-way skyline-join ability to process skyline-join queries over more than two data sources. Multi-way skyline-join queries are useful for scenarios in which the attributes used in the skyline query are stored in more than two disjoint tables and the skyline is defined over the result of a join operation over these tables. This occurs especially in multi-criteria decision applications that need to integrate information from multiple datasets.

Example 6. *Figure 3.14 shows an example application that leverages a* multi-way skyline-join *query to produce meaningful results for decision-making. This example builds on the two-way skyline-join example shown in Figure 3.1(b). In addition to the* Restaurants *and* Bars *tables, this example also includes a* Movies *table that stores information related to movies released in different cities. As before, the attributes used in the skyline query come from different sources and the skyline is defined over the result of join operations performed between these tables. In this example, users are interested in finding the best nearby <movie, restaurant, bar> triples, which include movies that have a high rating and have been reviewed by many people, restaurants that close late at night and have a high star-rating, and bars that are of high quality and offer low prices.* ◇

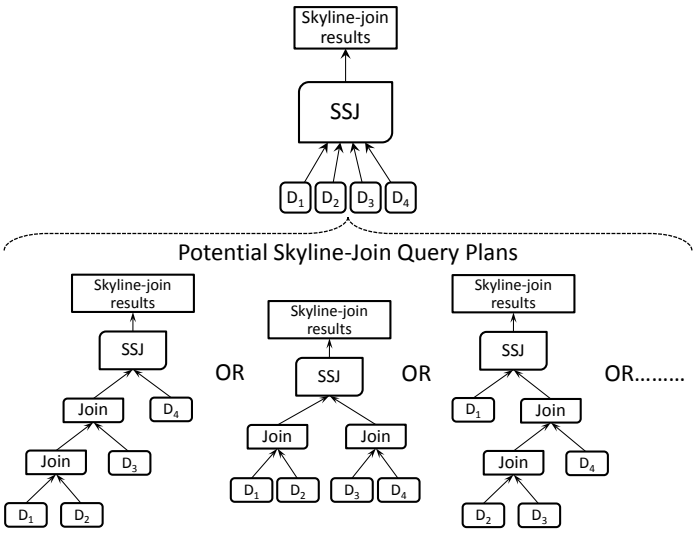


Figure 3.15: A naive approach: using binary SSJ to answer multi-way skyline-join queries

3.4.1 A First Attempt

One way to leverage the proposed binary SSJ algorithms to process multi-way skyline-join queries is to use SSJ only at the last step of the process. For instance, Figure 3.15 shows query plans that first carry out a join between datasets at the lower level of the plan, and these join results are pipelined through to the SSJ operator at the last step to obtain the final skyline-join result. This naive approach, however, faces many challenges.

One of the key challenges of executing skyline-join queries on multiple data sources is query planning. Figure 3.15 shows an example SSJ operation on four input data sources (D_1, D_2, D_3, D_4). As illustrated in this figure, the number of query plans to consider can be potentially very high. The task of enumerating these plans can be time-consuming. Moreover, even if we can enumerate all these plans, it is not immediately clear how to pick the best query plan among the many alternatives. In fact, since a large number of join operations have already been implemented among all the data sources except one, the savings provided by the final $\mathtt{S}^2\mathtt{J}$ or $\mathtt{S}^3\mathtt{J}$ operation

Algorithm 5: $\texttt{S}^2\texttt{J-M}(\mathcal{D}_{out}, D_{in}, A_S, A_J, T)$

Input:

$\mathcal{D}_{out}$: Outer set, D_{in}: Inner table, A_S: Set of skyline attributes of $\mathcal{D}_{out}$ and D_{in}, A_J: Join attribute set

T: Trie maintained for D_{in}, R: Set of regions to be examined during join-based skyline processing

$ds(L_i)$: Dominance score of current layer in $\texttt{TopK}(\mathcal{D}_{out})$

Output:

Skyline result S produced by $\texttt{TopK}(\mathcal{D}_{out})$ $\texttt{SSJ}_{A_J,A_S}$ D_{in}

Procedure:

Initialize T;

for each layer $L_i \in \texttt{TopK}(\mathcal{D}_{out})$ scanned from $L_1, L_2, \ldots, L_n$ **do**

if data point scanned is the first data point in L_1 or T has been changed **then**

R = RegionsToExamine(T, $ds(L_i)$);

/* Invoke the `RegionsToExamine` algorithm with parameters T and $ds(L_i)$ only if T has been changed, else R remains the same for the next round. `RegionsToExamine` is also invoked once at the beginning when the first data point in L_1 is scanned */

end if

initialize set S_i, the set of skyline points for L_i;

/* Contains a set of points that do not dominate each other */

for each $r \in R$ **do**

/* r is the Z-value region prefix */

minZval(r) = $minZ(r)$;

maxZval(r) = $maxZ(r)$;

/* Minimum and maximum Z-values of r are obtained */

$joinSet = L_i \bowtie_{A_J} D_{in}$ in the region defined by [minZval(r), maxZval(r)];

add the results $\in joinSet$ to S_i such that only skyline points are obtained in S_i;

for each skyline point $s = s_{out}||s_{in} \in S_i$ **do**

$U = min(s_{out})$;

T = Rmarker(s_{in}, T, U);

/* Update T based on $s_{in} \in D_{in}$ and the bound U obtained by taking the minimum of the coordinate values of $s_{out} \in \mathcal{D}_{out}$ */

end for

end for

add S_i to S such that only skyline points are obtained in S;

end for

return S

Figure 3.16: The $\texttt{S}^2\texttt{J-M}$ algorithm

are not likely to be significant. To avoid these difficulties, in this section, we propose an M-way *layer/region pruning* (*LR-pruning*) technique and discuss how $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ can be extended to multi-way skyline-joins using this technique.

3.4.2 M-Way Version of the $\texttt{S}^2\texttt{J}$ *Algorithm (*$\texttt{S}^2\texttt{J-M}$*)*

A detailed pseudocode of $\texttt{S}^2\texttt{J-M}$ is presented in Figure 3.16. At a higher level of abstraction, the algorithm works as follows: Given M data sources in set $\mathcal{D}$, split into an outer data set, $\mathcal{D}_{out}$, and an inner table, D_{in}, a set of join attributes, A_J, and a set of skyline attributes, A_S, the $\texttt{S}^2\texttt{J}$ algorithm over M data sources, i.e. $\texttt{S}^2\texttt{J-M}$, proceeds as follows:

1. $S_{all} = \emptyset$
2. $\texttt{S}^2\texttt{J-M}$ scans $\texttt{TopK}(\mathcal{D}_{out})$ from layer L_1 to L_n.
3. For each layer L_i:
   (a) $\texttt{S}^2\texttt{J-M}$ invokes the $\texttt{RegionsToExamine(T, }ds(L_i)\texttt{)}$ function to obtain the corresponding set of Z-value regions, R_i, from the trie structure, T, maintained for D_{in}. The Z-value regions of D_{in} obtained in this step will participate in the skyline-join process with L_i.
   (b) $C = \emptyset$
   (c) For each region $r \in R_i$:
      - $C = C \cup (L_i \bowtie_{A_J} r)$ is carried out to combine the tuples of $\texttt{TopK}(\mathcal{D}_{out})$ in L_i with the tuples of D_{in} in r.
   (d) The skyline set, S_i, of $C \cup S_{all}$ is obtained.
   (e) $\texttt{Rmarker}(S_i)$ is invoked to mark the appropriate regions of D_{in} based on S_i (see Section 3.4.2).

(f) $S_{all} = S_{all} \cup S_i$

4. $\texttt{S}^2\texttt{J-M}$ proceeds until all the layers produced by $\texttt{TopK}(\mathcal{D}_{out})$ are processed or the entire dataset in $\mathcal{D}_{in}$ is pruned.

Note that the algorithm is similar to the two-way version of $\texttt{S}^2\texttt{J}$ (presented in Figure 3.3) except that it relies on a novel M-way *LR-pruning* strategy to efficiently process skyline-join queries over more than two data sources. In particular,

- a novel, $\texttt{TopK}(\mathcal{D}_{out})$, operation is used for constructing the layers $L_1, L_2, \ldots, L_n$ of the outer data set, and
- the skyline result, S, is produced by $\texttt{TopK}(\mathcal{D}_{out})$ $\texttt{SSJ}_{A_J,A_S}$ D_{in}.

We next describe the M-way *LR-pruning* strategy and the changes it implies in the way the operator is implemented.

M-Way Layer/Region Pruning

In Section 4.4, we proposed two-way skyline-join algorithms, namely $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$, that leverage a novel *layer/region pruning* (*LR-pruning*) strategy to obtain skyline-join results in an efficient manner. *LR-pruning* relies on (1) the monotone ordering of the tuples in the outer table into *layers* of dominance, and (2) the clustering of the inner table tuples into *regions* based on the Z-values of the skyline attributes to support block-based pruning. Here, we propose an extension of *LR-pruning* strategy for the situations where we are given more than two data sources.

Let $\mathcal{D} = \{D_1(a_{1,1}, \ldots, a_{1,d_1}), D_2(a_{2,1}, \ldots, a_{2,d_2}), \ldots, D_M(a_{M,1}, \ldots, a_{M,d_M})\}$ represent the set of M data sources. In the M-way version of *LR-pruning*, the set $\mathcal{D}$ is first split into two: an *outer* set ($\mathcal{D}_{out}$) and an *inner* table (D_{in}), such that $\mathcal{D}_{out} \cup \{D_{in}\} = \mathcal{D}$.

Combined Dominance Layering of Outer Data Sources One of the key requirements of *LR-pruning* is the monotone ordering of the inputs into *dominance layers*. We meet this constraint by combining the data sources in the outer set, $\mathcal{D}_{out}$, into *layers* of dominance using a top-k join operator (Figure 3.17).

Top-k join algorithms such as *FA (Fagin's Algorithm)* [22], *TA (Threshold Algorithm)* [23], to name a few, assume that the tuples in each table being joined are sorted based on a table-specific scoring function. Given a (monotonic) merge function that can be used for combining these table-specific scores, the algorithms identify the k join tuples with the highest score, without having to scan all data tables. Most importantly, these algorithms tend to generate their results progressively, in that top-k join results can be obtained before the $(k+1)^{th}$ top join result is obtained. We leverage the algorithms to efficiently combine the tuples in the outer set, $\mathcal{D}_{out}$, into combined *layers* of dominance.

Let $A_{S_{out}} \subseteq A_S$ denote the skyline attributes that come from the data sources in $\mathcal{D}_{out}$. We use the TA algorithm [23] to obtain the top-k join results, $\texttt{TopK}(\mathcal{D}_{out})$, of the data sources in $\mathcal{D}_{out}$ sorted in the descending order of the overall scores of the combined tuples from $\mathcal{D}_{out}$. The scores are calculated using the MAX monotone aggregate function applied to $A_{S_{out}}$, since MAX dominance layers support skyline queries based on the MAX preference function. As mentioned earlier in Section 3.3.1, each layer L_i has a dominance score, $ds(L_i)$, defined as

$$ds(L_i) = max\Big(p.a \mid (a \in A_{S_{out}}) \wedge (p \in L_i)\Big).$$

The dominance layers obtained through $\texttt{TopK}(\mathcal{D}_{out})$ are in descending order of the value of $ds(L_i)$.

Relying on the progressive nature of the TA algorithm, the layers are generated and accessed from the most dominant (L_1) to the least dominant (L_n) layer by start-

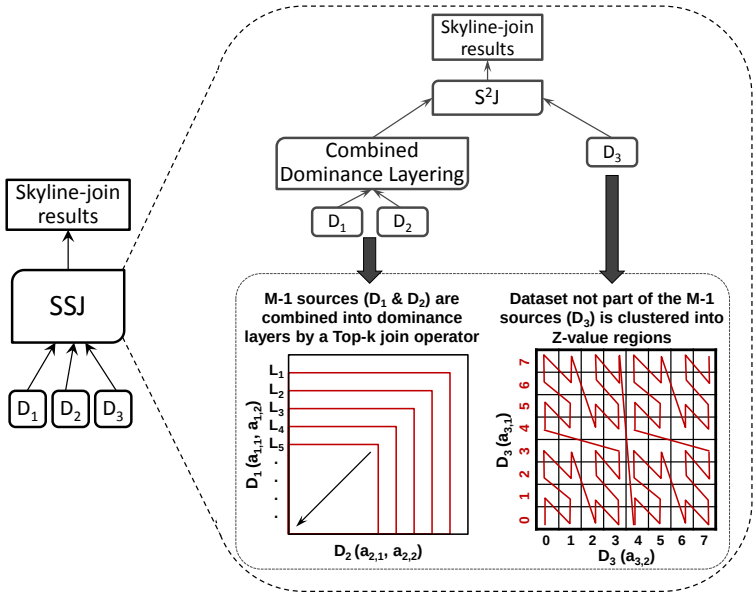


Figure 3.17: Layer/region organization of multiple datasets

ing with $k = 1$ and increasing progressively increasing k to generate more layers. Intuitively, $\texttt{TopK}(\mathcal{D}_{out})$ generates layer L_1 by producing all top-1 join results with the same $ds(L_1)$ value, the second layer L_2 is generated by producing the next set of top results with the value, $ds(L_2)$ $(< ds(L_1))$, and so on.

Example 7. *Figure 3.17 illustrates our proposed idea. In this example,* $\mathcal{D}_{out} = \{D_1, D_2\}$ *and* $A_{Sout} = \{a_{1,1}, a_{1,2}, a_{2,1}, a_{2,2}\}$. *The top-k join of the data sources in* $\mathcal{D}_{out}$ *(denoted as* $\texttt{TopK}(\mathcal{D}_{out})$*) results in* dominance layers *ordered on the overall score obtained by applying the MAX aggregate function to* A_{Sout}. *Here,* $\texttt{TopK}(\mathcal{D}_{out})$ *plays the role of the so-called* outer *table used in the LR-pruning strategy over pairs of data sources (Section 3.3.1). Note that, each dataset in* $\mathcal{D}_{out}$ *is presorted in the descending order of the MAX of their skyline attributes before being sent to the top-k join operator. This serves as the sorted access required by the TA algorithm. For instance, in Figure 3.17,* D_1 *is sorted in the descending order of* $max(a_{1,1}, a_{1,2})$ *before it is fed into the top-k join operator. Similarly,* D_2 *is sorted in the descending order of* $max(a_{2,1}, a_{2,2})$. ◇

Region Organization of the Inner Table The tuples in the inner table D_{in} are mapped onto a Z-order curve based on the corresponding skyline attribute set, $A_{Sin} \subseteq A_S$, as described in Section 3.3.1.

Marking the Inner Table based on Outer Layers

Similar to the two-source version of $\texttt{S}^2\texttt{J}$, the above-mentioned M-way version of $\texttt{S}^2\texttt{J}$ maintains a *trie*-based data structure on D_{in} to keep track of the so-called *Not-dominated (ND)*, *Dominated for U (DOM U)* and *Partially-Dominated (PD)* regions of D_{in} *relative to the layers* produced by $\texttt{TopK}(\mathcal{D}_{out})$. As $\texttt{S}^2\texttt{J-M}$ discovers new skyline points that can *prune regions* in D_{in}, it calls the `Rmarker` book-keeping function (Figure 3.4) that marks the regions of D_{in} as described in Section 3.3.2 with one important exception – the calculation of the bound U when marking a region *Dominated for U (DOM U)*.

Let S_i be the set of skyline points of the outer data set layer, L_i, and let $s = s_{out}||s_{in} \in S_i$ be a skyline point; s_{out} is obtained through $\texttt{TopK}(\mathcal{D}_{out})$ and corresponds to the outer data set, $\mathcal{D}_{out}$. As mentioned in Section 3.3.2, a node, v, that is originally marked as *Partially-Dominated* (PD) or *Not-Dominated* (ND) would become dominated if an s_{in} dominating this region is discovered. Let $S_{in}(v)$ be the set of skyline points in D_{in} that dominates node v and $S_{out}(v)$ be the corresponding points in $\texttt{TopK}(D_{out})$. Then, the node v will be marked as DOM U, where U is

$$U = \max_{s_{out} \in S_{out}(v)} \Big(\min_{a \in A_{S_{out}}} (s_{out}.a) \Big).$$

Here, A_{Sout} represents the skyline attributes of $\mathcal{D}_{out}$ and the bound U is now based on $M-1$ data sources that are combined through a top-k operation, rather than a single data source as in the case of `Rmarker` described in Section 3.3.2.

Correctness

The correctness of the S²J-M algorithm follows directly from the correctness of the S²J algorithm and the monotonically non-increasing (i.e., layered) order of the results produced by the top-k join algorithm used for combining the data sources in $\mathcal{D}_{out}$.

Time Complexity

In Section 3.3.2, we have seen that the time complexity of the S²J operator consists of two major components: the time, $cost_{prep}$, needed to prepare the dominance layers of the outer table and relevant data structures on the inner table and the time, $cost_{comp}$, needed to compute the skyline join results.

In the case of S²J-M, sorting and B-tree creation needs to be done for each of the data tables in the outer set $\mathcal{D}_{out}$, leading to the overall data preparation cost of

$$cost_{prep} = O\left(\left(\sum_{D \in \mathcal{D}_{out}} \underbrace{(|D| \times log(|D|)}_{sorting}\right) + \underbrace{\left(|D_{in}| \times log(|D_{in}|) + \sum_{h=0}^{log_f\ (|D_{in}|)} \frac{|D_{in}|}{f^h}\right)}_{indexing}\right),$$

where $|D|$ is the size of the table D.

A key difference of S²J-M from the S²J algorithm is that, during skyline-join computation process, the outer layers of the data are not available to S²J-M in a precomputed manner, but the scan that gives these layers is performed through a top-k join algorithm which incrementally combines the $M-1$ data sources in $\mathcal{D}_{out}$. Therefore, $cost_{comp}$ for S²J-M differs from the cost formula presented in Section 3.3.2 only in that the scan cost, $cost_{scan}(n)$, needs to be replaced with the cost, $cost_{TA}(\mathcal{D}_{out})$, of combining data in $\mathcal{D}_{out}$ using the TA algorithm [23]:

$$cost_{TA}(\mathcal{D}_{out}) + \sum_{c=1}^{\#\ outer\ layers} cost_{INLJ}\Big(ls(L_c), nip\big(l, ds(L_c)\big)\Big),$$

where (a) $cost_{TA}(\mathcal{D}_{out})$ is the cost of combining data in $\mathcal{D}_{out}$ using the TA algorithm [23], (b) # *outer layers* is the number of layers in the outer relations, (c) $ls(L_c)$ is the number of tuples in the outer layer, L_c, (d) $ds(L_c)$ is the dominance score corresponding to outer layer, L_c, (e) $nip(l, B)$, is the number of inner pages in D_i that are either not dominated yet or dominated with a bound B given the trie with depth l, and (f) $cost_{INLJ}(x, p)$ is the cost of performing index nested loop join of x tuples in the outer relation, with p unpruned pages in the inner table.

Space Complexity

Like S²J, the S²J-M algorithm has to maintain two distinct data structures for the inner table: a B-tree that supports efficient candidate enumeration and a trie that maintains the dominance markings. This leads to the same worst-case complexity as that of S²J:

$$cost_{space} = \underbrace{O\Big(\prod_{D_i \in \mathcal{D}} |D_i|\Big)}_{Candidates} + \underbrace{O\Big(\sum_{h=0}^{log_f(|D_{in}|)} \frac{|D_{in}|}{f^h}\Big)}_{Btree} + \underbrace{O(min\{2^t, |D_{in}|\})}_{trie},$$

where $|D|$ is the number of tuples in data set, $D \in \mathcal{D}$, f is the fanout of the Btree and, t is the bound on the height of the trie.

Note that since the outer table is processed one layer at a time (and also thanks to the group skyline optimization discussed in Section 3.4.4), the number of candidates that need to be maintained at a given point time is, in practice, much smaller.

Selecting the Outer Set and the Inner Table

As stated before, the subset $\mathcal{D}_{out}$ contains $M - 1$ data sources chosen from the set $\mathcal{D}$, except the data source selected as the inner table. Hence, the number of possible *skyline-sensitive join (*SSJ*)* query plans is guaranteed to be linear (M) in terms of

the number of data sources. As we have already seen in Section 3.3.2, the query plans for $\texttt{S}^2\texttt{J}$ should be selected such that the inner table is not anti-correlated to promote pruning. Moreover, as we experimentally verify in Section 3.5.4, query plans that contain at least one correlated data source in their outer sets are more suitable to skyline-join processing. This is because, correlated data sources in the outer set helps prune false-positives much faster and earlier, without having to scan the skyline-join candidate list very deep. More specifically, the bounds used by LR-pruning to mark the inner table are higher when the outer set contains correlated data sources, leading to higher pruning opportunities. Lastly, we would recommend picking a SSJ query plan that has data sources with smaller cardinalities in its outer set, so that the amount of data scanned by the $\texttt{S}^2\texttt{J-M}$ algorithm is minimized.

3.4.3 M-Way Version of the $\texttt{S}^3\texttt{J}$ *Algorithm (*$\texttt{S}^3\texttt{J-M}$*)*

One potential disadvantage of the $\texttt{S}^2\texttt{J-M}$ algorithm described above is that it needs to combine all data in $\mathcal{D}_{out}$ using a top-K join algorithm, like TA algorithm [23]. Naturally, when the data sources in the outer set are large, this step itself may be a significant cost. In contrast, like the $\texttt{S}^3\texttt{J}$ algorithm described in Section 3.3.3, the multi-way (M-way) version of the $\texttt{S}^3\texttt{J}$ algorithm, called $\texttt{S}^3\texttt{J-M}$, implements an early stopping condition and, therefore, avoids the need to scan any of the input tables entirely in order to obtain the skyline-join results.

The $\texttt{S}^3\texttt{J}$ algorithm, described in Section 3.3.3, processes skyline-join queries over a pair of data sources (called outer table and inner table) by repeatedly swapping the roles of the outer and inner data tables. The $\texttt{S}^3\texttt{J-M}$ algorithm also follows a similar strategy: Let $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_M\}$ represent the set of M possible SSJ query plans. Here, a query plan Q_i is given by $Q_i = \langle \texttt{TopK}(\mathcal{D}_{out.i})\ \texttt{SSJ}_{A_J, A_S}\ D_{in.i} \rangle$, where $\mathcal{D}_{out.i}$ represents the *outer* set and $D_{in.i}$ is the *inner* table for query plan Q_i. The $\texttt{S}^3\texttt{J-M}$

Algorithm 6: $\texttt{S}^3\texttt{J-M}(\mathcal{D}_{out}, D_{in}, A_S, A_J)$

Input:

$\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_M\}$: A set of M SSJ query plans, where $Q_j = \langle \texttt{TopK}(\mathcal{D}_{out.j})\ \texttt{SSJ}_{A_J, A_S}\ D_{in.j} \rangle$)

$\mathcal{D}_{out.j}$: Outer set for query plan Q_j, $D_{in.j}$: Inner table for query plan Q_j

A_S: Set of skyline attributes, A_J: Join attribute set, $T_{in.j}$: Trie maintained for $D_{in.j}$

R: Set of regions to be examined during join-based skyline processing

$ds(L_i)$: Dominance score of current layer in $\texttt{TopK}(\mathcal{D}_{out.j})$

Output:

Skyline result S produced by $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_M\}$

Procedure:

Initialize $T_{in.j}$ $\forall D_{in.j} \in \mathcal{Q}$;

for each query plan $Q_j \in \mathcal{Q}$ **do**

read an element e in layer $L_{i,j} \in \texttt{TopK}(\mathcal{D}_{out.j})$;

$p(L_{i,j}) = \langle [ds(L_{i,j}), \ldots, ds(L_{i,j})], [ds(L_{i,j}), \ldots, ds(L_{i,j})], \ldots, [ds(L_{i,j}), \ldots, ds(L_{i,j})] \rangle$;

if not all $M-1$ subpoints in $p(L_{i,j})$ are marked as DOM U **then**

$S_{i,j} = \texttt{S}^2\texttt{J-M}(e(L_{i,j}), D_{in.j}, A_S, A_J, T_{in.j})$;

/* $S_{i,j}$ is the skyline result set for $L_{i,j} \in \texttt{TopK}(\mathcal{D}_{out.j})$ returned by $\texttt{S}^2\texttt{J-M}$ */

else

check the stopping condition: $\bigwedge_{h=1,\ldots,M,\ h \neq j} U_{max,h} \geq ds(L_{y,h})$;

/* $U_{max,h}$ is the largest bound in the corresponding *trie* for $D_{in.h} \mid h = 1, \ldots, M \ \wedge \ h \neq j$ for which each subpoint in $p(L_{i,j})$ is dominated in the trie $T_{in.h}$ */

if the stopping condition holds **then**

stop scanning the layers from $\texttt{TopK}(\mathcal{D}_{out.j})$ and stop using query plan Q_j;

end if

end if

add $S_{i,j}$ to S such that only skyline points are obtained in S;

end for

return S

Figure 3.18: $\texttt{S}^3\texttt{J-M}$ swaps the outer set and inner table in a round-robin manner

algorithm (Figure 3.18) executes each of the steps in $\texttt{S}^2\texttt{J-M}$ by first utilizing Q_1 as the query plan, then Q_2 as the query plan, and so on in a round-robin manner, until all skyline-join results are computed (Figure 3.19). As the $\texttt{S}^3\texttt{J-M}$ algorithm cycles through the query plans in $\mathcal{Q}$, the sets $\mathcal{D}_{out}$ and D_{in} change and, as a result, the top-k operator sees a different set of data sources at every stage of the cycle. In other

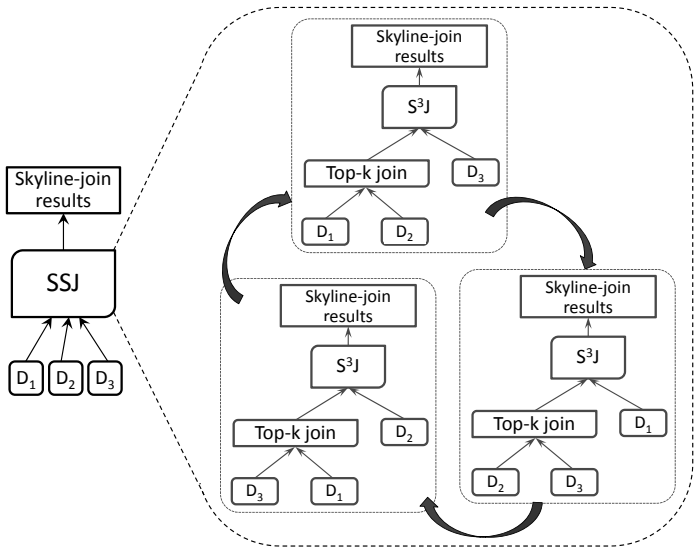


Figure 3.19: The S³J-M algorithm produces skyline-join results by cycling through all M possible SSJ query plans in a round-robin manner

words, S³J-M generates skyline-join results over M data sources by cycling through all M S²J query plans – the algorithm moves from the current plan, Q_i, to the next once it processes the tuples in the current layer, $L_{curr,i}$, of $\texttt{TopK}(\mathcal{D}_{out.i})$.

One key outcome of cycling through all the query plans in $\mathcal{Q}$ is that S³J-M may not have to scan all M datasets completely in order to obtain the skyline-join results. This is unlike the S²J-M algorithm, which fully scans the data sources in the *outer* set, $\mathcal{D}_{out}$, to compute the skyline results. Hence, S³J-M may stop without fully scanning any of the data sources. To achieve this, it utilizes an early stopping condition.

Early Stopping Condition

Let $Q_i = \langle \texttt{TopK}(\mathcal{D}_{out.i})\ \texttt{SSJ}_{A_J,A_S}\ D_{in.i} \rangle \in \mathcal{Q}$ be the current query plan being utilized by the S³J-M algorithm. $\mathcal{D}_{out.i}$ is the current set of data sources that serves as the outer set, $\mathcal{D}_{out}$, and $D_{in.i}$ represents the corresponding current inner table. Let also

- $L_{curr,i}$ denote the current layer being scanned in $\texttt{TopK}(\mathcal{D}_{out.i})$ of Q_i, and
- $L_{last,j}$ denote the last layer scanned in $\texttt{TopK}(\mathcal{D}_{out.j})$ for $Q_j \in \mathcal{Q} - \{Q_i\}$.

The early stopping condition is checked as follows:

1. For each data source $D_h \in \mathcal{D}_{out.i}$, the $\texttt{S}^3\texttt{J-M}$ algorithm

   (a) first constructs an extremum point, $p(L_{curr,i}, h)$,

   $$p(L_{curr,i}, h) = \Big\langle \underbrace{ds(L_{curr,i}), \ldots, ds(L_{curr,i})}_{number\ of\ skyline\ attributes\ of\ D_h} \Big\rangle,$$

   where $ds(L_{curr,i})$ is the largest attribute value for all skyline attributes in each of the $M-1$ datasets in $\mathcal{D}_{out.i}$, and then

   (b) checks if this extremum point is in a region marked dominated for the data set D_h.

2. If for any data source, $D_h \in \mathcal{D}_{out.i}$, the extremum, $p(L_{curr,i}, h)$ is not in a region marked dominated in the corresponding data, then $L_{curr,i}$ is processed.

3. If, on the other hand, for all $M-1$ data sets, the extremum, is in a region marked dominated, then for all $D_h \in \mathcal{D}_{out.i}$,

   (a) $\texttt{S}^3\texttt{J-M}$ finds, in the corresponding trie, the largest bound $U^{max,h}$ for which the extremum $p(L_{curr,i}, h)$ is dominated, and

   (b) checks if

   $$U_{max,h} \geq ds(L_{last,h}).$$

   If this condition holds for all $D_h \in \mathcal{D}_{out.i}$, then this means that the layer $L_{curr,i}$ and all the subsequent layers obtained from $\texttt{TopK}(\mathcal{D}_{out.i})$ can be eliminated for further consideration for query plan, Q_i.

Correctness

The correctness of the S^{3}J-M algorithm follows directly from the correctness of the S^{2}J algorithm and the monotonically non-increasing (i.e., layered) order of the results produced by the top-k join algorithm used for combining the data sources in $\mathcal{D}_{out}$.

Time Complexity

As before, the time complexity of the S^{3}J-M algorithm consists of two major components: the time, $cost_{prep}$, needed to prepare the dominance layers of the outer table and relevant data structures on the inner table and the time, $cost_{comp}$, needed to compute the skyline join results.

The data preparation cost, $cost_{prep}$, is similar to the corresponding cost of S^{2}J-M, except that all M data sources have to be sorted (not $M-1$ data sources as in S^{2}J-M) and the B-tree and trie need to be created for all M data tables:

$$cost_{prep} = O\left(\sum_{D\in\mathcal{D}}\left(\underbrace{(|D|\times log(|D|)}_{sorting} + \underbrace{\left(|D|\times log(|D|) + \sum_{h=0}^{log_f\ (|D|)}\frac{|D|}{f^h}\right)}_{indexing}\right)\right),$$

where $|D|$ is the size of the table D. On the other hand, since it repeatedly swaps query plans, the $cost_{comp}$ complexity of S^{3}J-M algorithm can be derived as

$$\sum_{i=1}^{M}\left(cost_{TA}(\mathcal{D}_{out.i}, k_i) \ \ + \sum_{c=1}^{k_i} cost_{INLJ}\Big(ls(L_c), nip\big(l, ds(L_c)\big)\Big)\right),$$

where (a) $cost_{TA}(\mathcal{D}_{out.i})$ is the cost of combining data in $\mathcal{D}_{out.i}$ using the TA algorithm [23], (b) k_i is the number of layers scanned for $\mathcal{D}_{out.i}$ before the early stopping condition becomes true for this Q_i, (c) $ls(L_c)$ is the number of tuples in the outer layer, L_c, (d) $ds(L_c)$ is the dominance score corresponding to outer layer, L_c, (e) $nip(l, B)$, is the number of inner pages in D_i that are either not dominated yet or

dominated with a bound B given the trie with depth l, (f) $cost_{INLJ}(x, p)$ is the cost of performing index nested loop join of x tuples in the outer relation, with p unpruned pages in the inner table. Consequently, the effectiveness of the S^{2}J-M algorithm depends both on how small k_i are (i.e., how early stopping conditions are enabled) and how well the tables pruned (which impacts both the cost of the index nested loop joins as well as how quickly stopping conditions are triggered).

Space Complexity

Like S^{3}J, the S^{3}J-M algorithm has to maintain a B-tree that supports efficient candidate enumeration and a trie that maintains the dominance markings. This leads to the worst-case

$$cost_{space} = \underbrace{O\Big(\prod_{D_i \in \mathcal{D}} |D_i|\Big)}_{Candidates} + O\Bigg(\sum_{D_i \in \mathcal{D}} \underbrace{\Big(\sum_{h=0}^{log_f(|D_i|)} \frac{|D_i|}{f^h}\Big)}_{Btree} + \underbrace{(min\{2^t, |D_i|\})}_{trie}\Bigg),$$

where $|D|$ is the number of tuples in data set, $D \in \mathcal{D}$, f is the fanout of the Btree and, t is the bound on the height of the trie. As mentioned earlier, since the outer data set is processed one layer at a time and thanks to the group skyline optimization discussed next, the number of candidates that need to be maintained at a given point time in memory is, in practice, much smaller.

3.4.4 Further Optimizations

In this section, we present two additional optimization techniques that together help improve the performance of the S^{2}J-M and S^{3}J-M algorithms.

OPT 1: Group Skyline Optimization

As a preprocessing step, we can group the tuples of each data source $D_i \in \mathcal{D}$ by the join attribute, A_J, and run a known single-source skyline algorithm, such as [13], on

these groups to obtain the skyline set of each group. As discussed in [78], tuples that are not in the group skyline cannot generate any skyline-join results, when joined with tuples from the other relation. `S²J-M` and `S³J-M` implement the join operation needed to generate skyline-join candidates by using only the group skyline tuples of the outer data sources in $\mathcal{D}$. It is important to note that if all the join attribute values of a dataset $D_i \in \mathcal{D}$ are unique, then this optimization technique becomes obsolete for D_i. Note that, this optimization does not affect the correctness of `S²J-M` and `S³J-M` even when the group skylines are computed for each table in the outer set separately. This is because, if any tuple, s, from a table in the outer set, $\mathcal{D}_{out}$, is dominated by another tuple, s', with the same join attribute value, then the join tuples produced using s are guaranteed to be dominated by the join tuples generated using s'.

OPT 2: Monotonic Inner Table Optimization

Let $A_{Si} \subseteq A_S$ denote the skyline attributes of the dataset $D_i \in \mathcal{D}$ and let $d_i = |A_{Si}|$ represent the number of skyline attributes in D_i. As an additional preprocessing step, we can sort the data sources contributing to the inner table in the descending order of the product ($P(t) = \prod_{h=1}^{d_i} t.a_h$) or sum ($S(t) = \sum_{h=1}^{d_i} t.a_h$) of the skyline attributes in D_i. With this optimization, when `S²J-M` and `S³J-M` process the tuples in a particular layer, $L_{curr,j}$, of the outer set, $\mathcal{D}_{out}$, they do so in the descending order of the product (or sum) of the skyline attributes in the corresponding inner table, D_{in}. The higher an inner tuple t's score (given by $P(t)$ or $S(t)$) the more tuples it is likely to dominate in the inner table, D_{in}. Hence, high-scoring tuples will discard more tuples in D_{in} per layer, $L_{curr,j}$, of the outer set, $\mathcal{D}_{out}$. With this sorting, `S²J-M` and `S³J-M` may discard tuples more effectively in D_{in} than before. Thus, processing joins and materializing candidates in the monotonic order of the tuples in the inner table, D_{in}, may be useful in further reducing wasted work in terms of join operations and dominance checking.

Table 3.1: Summary of parameters used in the experiments

Parameter	Values (Default values are in bold)
Data distribution	Correlated, **Independent**, Anti-Correlated
Correlation Coefficient	0.63, **0.73**, 0.86, 0.96
Cardinality per dataset (n)	10K, **100K**, 1000K
Number of join attributes (j)	**1**
Number of skyline attributes per dataset (d)	**2**, 3, 4
Total number of skyline attributes (s)	**4**, 6, 8
Join rate (r)	0.0001, 0.001, 0.01, 0.1, 1, **10**
Maximum trie depth (l)	1, 2, 3, 4, **5**, 6, 7, 8, 9, 10
Number of datasets (M) for M-way join	**3**, 4, 5, 6

Note that only S^{3}J-M requires that all data sources in $\mathcal{D}$ be sorted on one of the above-defined monotone scoring functions. This is because each dataset $D_i \in \mathcal{D}$ plays the role of the inner table, D_{in}, at some point during the execution of S^{3}J-M. The S^{2}J-M algorithm, on the other hand, needs only the dataset chosen to be in the inner table, D_{in}, to be sorted.

3.5 Experimental Evaluations

In this section, we evaluate the efficiency and effectiveness of the two-way and M-way versions of S^{2}J and S^{3}J by varying various key parameters and by comparing them against alternative schemes, including iterative skyline-join [68], PrefJoin [36], and the SFSJ-RR and SFSJ-SC algorithms [78].

3.5.1 Experimental Setup

Our evaluations were conducted on a setup with an Intel Core 2 Duo 2.33GHz processor, 2GB RAM, and Windows 7 operating system. The algorithms presented

Table 3.2: Data sizes of data sources used in Figure 3.21 (stored in MySQL database)

Number of Tuples per Dataset	Data size per Dataset
10K tuples	0.42 MB
100K tuples	4.52 MB
1000K tuples	38.58 MB

in this chapter were implemented in Java and the experiments were run on data that was stored locally. For comparison purposes, the Java implementations of the SFSJ-RR and SFSJ-SC algorithms were obtained from the authors of [78]. In addition, we implemented PrefJoin [36] and the iterative skyline-join algorithm [68] to make further comparisons. The B-tree index is built using Berkeley DB Java Edition 3.3.87[4]. The trie implementation in Java was adapted from `http://www.technicalypto.com/2010/04/trie-in-java.html`.

To compare M-way skyline-join algorithms, we have implemented the conventional skyline-join approach, in which we completely join the relevant data sources in order to materialize all candidate tuples, and then apply an existing single-source skyline algorithm to obtain the skyline-join results. The single-source algorithms used in our implementations of conventional skyline-join include *SFS* [15] and *bitmap* skyline [70].

• **Datasets.** The evaluations were carried out on over 30 different synthetic datasets and 4 different real/benchmark datasets. Synthetic datasets as described in [13] were generated based on correlated, anti-correlated, and independent distributions[5]. The cardinality of the datasets (n) was varied between 10,000 and 1 million tuples per data source. Table 3.2 lists the data sizes (MB) of the data sources used in Figure 3.21.

[4]Downloaded from `http://www.oracle.com/technology/software/products/berkeley-db/je/index.html`.

[5]Random dataset generator available for download at `http://randdataset.projects.postgresql.org/`.

The data is stored in a MySQL database.

The join rates (r) considered were 0.0001 (i.e., 1 in 10,000 data tuples in a data source joins a tuple in another data source), 0.001, 0.01, 0.1, 1 (i.e., all join attribute values in a particular dataset are unique), and 10 (i.e., each join attribute value in a data source repeats 10 times). The dimensionality (d) of the skyline attribute set of each data source was varied between 2 and 4, hence the dimensionality of the result set ($s = |A_S|$) obtained after D_{out} $\texttt{SSJ}_{A_J,A_S}$ D_{in} varies between 4 and 8. The number of data sources (M) for the $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$ algorithms was varied between 3 and 6.

In addition to these synthetic datasets, we also used the NBA[6] and the TPC-H benchmark[7] datasets. Table 3.1 summarizes the various parameters used in the experiments; the default values are shown in bold.

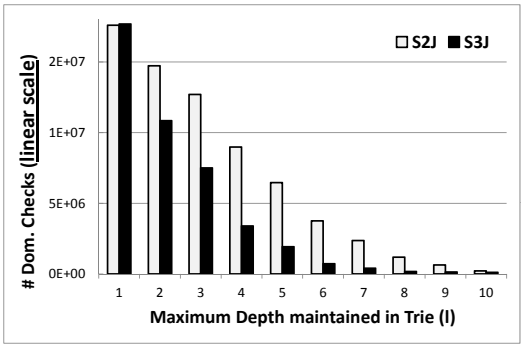
• **Evaluation Measures.** As is common in assessing skyline algorithms, we used execution time as the major metric in evaluating our methods. Execution time is the duration from the time an algorithm starts to the time it returns the entire skyline set. In addition, we used the total number of tuples scanned (sum of the depths of tuples accessed from the two input tables), the number of dominance checks and the number of join results as other evaluation metrics.

Both our approaches and the competitor approaches require the inputs to be sorted. Since, as experimentally verified in Section 3.5.2, the time to create the required data structures is small compared to the time needed to create and consider skyline-join candidates, all indices and sorted records are prepared prior to running the experiments.

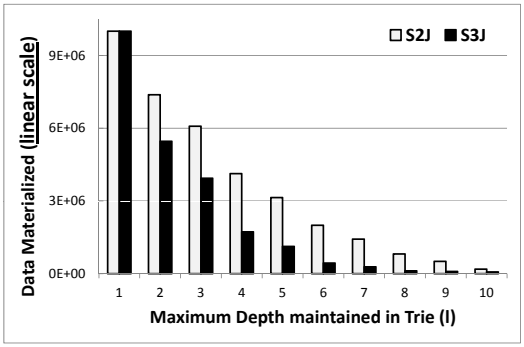
Unless otherwise specified, each experiment is run five times and the results reported are the averages of the five runs.

[6]Available at `http://skyline.dbai.tuwien.ac.at/datasets/nba/`.

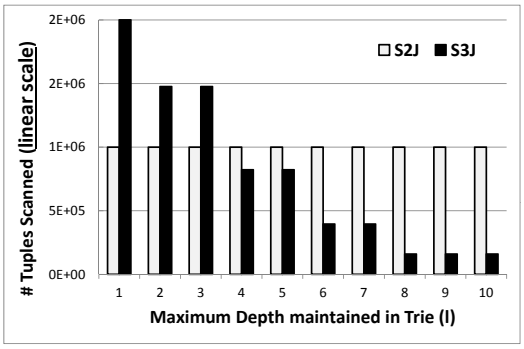
[7]TPC-H generator from `http://www.tpc.org/tpch/default.asp`.



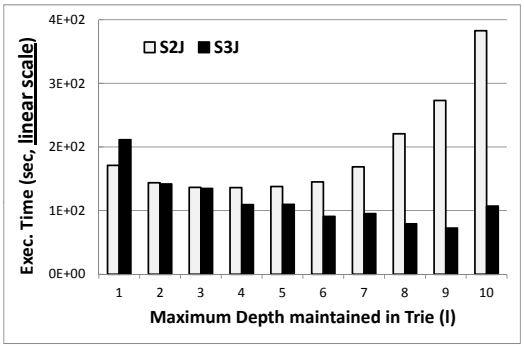
(a) Dominance checks



(b) Data materialized



(c) Tuples scanned



(d) Execution time

Figure 3.20: Effect of maximum depth (l) of the trie structure ($n = 1M/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$)

3.5.2 Analysis of $\mathsf{S^2J}$ *and* $\mathsf{S^3J}$

Before comparing $\mathsf{S^2J}$ and $\mathsf{S^3J}$ to other algorithms, we first experimentally analyze the performance characteristics of $\mathsf{S^2J}$ and $\mathsf{S^3J}$. In particular, we investigate the impact of the region granularity defined by trie depth l and the memory utilization of data structures.

Impact of l

If $l = 1$, the trie stores information about data points in only 2 blocks: one block that covers all data points with common region prefix 1 and another block that covers all data points with common region prefix 0. For a given l, the trie can store information

Table 3.3: Disk IOs performed during random accesses on B-tree index and sorting when $l = 5$ (Independent data source, $j = 1$, $d = 2$, $s = 4$, $r = 10$)

(a) Index Disk IOs for S²J

Data Cardinality	10K	100K	1000K
Cache Misses	8,733	85,363	838,372
(Randoms Reads)	(522)	(73,714)	(828,843)

(b) Index Disk IOs for S³J

Data Cardinality	10K	100K	1000K
Cache Misses	17,464	170,527	1,384,574
(Random Reads)	(991)	(141,215)	(1,346,008)

up to 2^l regions.

Figure 3.20 shows the effect of l on execution time, number of tuples scanned, number of intermediate candidates materialized, and the number of dominance checks[8]. As can be seen here, for both S²J and S³J, the number of dominance checks (Figure 3.20(a)) and the number of intermediate data points materialized (Figure 3.20(b)) decrease as l increases. As expected, while for S²J the outer table needs to be scanned in its entirety independent of l, S³J is able to better prune the number of scanned tuples when the granularity of the space is fine; i.e., l is large (Figure 3.20(c)).

Figure 3.20(d) shows the execution time behaviors of S²J and S³J. As can be seen here, the impact of l on the execution time is not monotonic: as l increases, the execution times first decrease and, after some point, start increasing. Thus, there is a trade-off between the maximum depth of the trie and the gains achieved in terms of execution time. Remember, from Section 3.3.2, that when the maximum depth of the trie is reached, the data in any "still non-dominated" regions need to go into

[8]Note that this experiment was carried on datasets with independent data distribution. Similar results were obtained for other datasets with different cardinality and dimensionality, and hence, these graphs are not shown.

Table 3.4: Disk IOs performed during random accesses on B-tree index and sorting when $l = 1$ (Independent data source, $j = 1$, $d = 2$, $s = 4$, $r = 10$)

(a) Index Disk IOs for $\texttt{S}^2\texttt{J}$

Data Cardinality	10K	100K	1000K
Cache Misses	10,138	101,160	1,011,420
(Randoms Reads)	(501)	(78,495)	(988,594)

(b) Index Disk IOs for $\texttt{S}^3\texttt{J}$

Data Cardinality	10K	100K	1000K
Cache Misses	20,282	202,304	1,698,191
(Random Reads)	(1,046)	(152,246)	(1,645,071)

Table 3.5: Disk IOs performed during random accesses on B-tree index and sorting when $l = 9$ (Independent data source, $j = 1$, $d = 2$, $s = 4$, $r = 10$)

(a) Index Disk IOs for $\texttt{S}^2\texttt{J}$

Data Cardinality	10K	100K	1000K
Cache Misses	8513	82,764	809,527
(Randoms Reads)	(528)	(71,076)	799,929

(b) Index Disk IOs for $\texttt{S}^3\texttt{J}$

Data Cardinality	10K	100K	1000K
Cache Misses	12,805	103,503	1,007,890
(Random Reads)	(916)	(86,860)	(975,384)

a join operation with data from the other relation. Therefore, a higher trie depth may increase the chances of finding dominated regions that can then be pruned away. Therefore, higher values of l can help in achieving faster execution times. However, we observed that beyond a certain value of l it actually becomes cheaper to materialize the skyline candidates through joins, rather than repeatedly checking for possibly pruned regions within a dense trie structure. To see why, consider that in the worst case, for the maximum possible value of l, the trie stores the pruned data point itself at the leaf level. This would mean that the pruned regions would no longer be

represented as blocks of data points, but instead they would be individual data points. Hence, checking for pruned regions will become as expensive, if not more expensive, as performing pairwise point-to-point dominance checks. For the $\texttt{S}^2\texttt{J}$ algorithm the turning point comes early, between $l = 3$ and 5, whereas the $\texttt{S}^3\texttt{J}$ algorithm benefits from better granularity until $l \sim 9$. This is because $\texttt{S}^3\texttt{J}$ is able to stop without having to scan all the layers in the datasets.

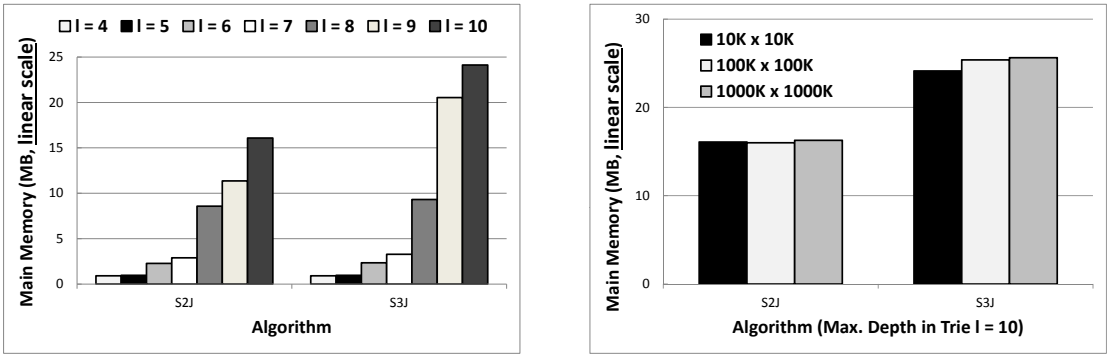
Tables 3.3, 3.4, and 3.5 show the disk IOs performed during random accesses on the B-tree index during candidate enumeration at different settings of the maximum trie depth, l. According to the Berkeley DB documentation[9], *Random Reads* is defined as the number of disk reads that required the repositioning of the disk head more than 1 MB from the previous file position, and *Cache Misses* is the total number of requests made for database objects that were not in memory.

As can be seen from the results, $\texttt{S}^3\texttt{J}$ performs more disk IOs than $\texttt{S}^2\texttt{J}$ because $\texttt{S}^3\texttt{J}$ swaps the outer and inner tables during the skyline-join process, and hence, has to make random accesses on both tables. Also, in general, the overall number of disk IOs falls for both $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ when the maximum trie depth is increased from $l = 1$ (Table 3.4) to $l = 9$ (Table 3.5). This is because at $l = 9$ the region markings are more specific, hence the inner table contains more regions that are marked as pruned. Note that as l increases, the disk IO drops faster for $\texttt{S}^3\texttt{J}$ than for $\texttt{S}^2\texttt{J}$, indicating that $\texttt{S}^3\texttt{J}$ benefits more from the finer pruning of the space.

Storage Requirements

In addition to storing the candidate and skyline tuples generated during the process, our algorithms need two supporting data structures: (a) a B-tree for indexing the

[9] `http://docs.oracle.com/cd/E17277_02/html/java/com/sleepycat/je/EnvironmentStats.html`



(a) (b)

Figure 3.21: Main memory utilization by the trie: (a) Effect of maximum depth, l ($n = 1000K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$), (b) Effect of cardinality, n, of datasets ($j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 10$)

data, and (b) a trie structure for maintaining the region markings.

The B-tree index, which is stored on disk and used for join processing, relies on a composite key formed by combining the join attribute and the Z-values of the set of skyline attributes to support simultaneous region pruning with join candidate search (Section 3.3.2). To assess the storage overhead of this combined key, we ran an experiment on a dataset with 1 million tuples ($j = 1$, $d = 2$, $s = 4$, $r = 10$) and found that for this dataset the Berkeley DB B-tree index built on only the join attribute occupied 40.3 MB of hard disk space, whereas an index built on the composite key formed by the join attribute and the Z-values occupied 96.9 MB of disk space. This rough doubling in space occupied is as expected as the join key and the Z-value each is stored as a Java integer of size 4 bytes.

Perhaps more important is the memory consumption of the in-memory trie data structure, which keeps track of the pruned and non-pruned regions of the data space. The storage overhead of the trie depends on the number of marked regions; in the worst case, the trie would need to maintain markings on each and every data point; this would mean that the pruned regions can no longer be represented as blocks of

Table 3.6: Time Cost of Sorting and Z-order based B-tree Indexing of a single data source (Independent data source, $d = 2$, $r = 10$)

Data Cardinality	10K	100K	1000K
Sorting	23.9 msec.	246.8 msec.	887.9 msec.
Indexing	1.1 sec.	2.8 sec.	30.4 sec.

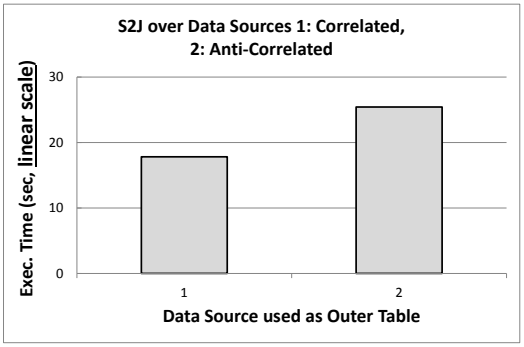
data points, but instead they would be individual data points. As shown by the experimental results in Figure 3.21, the memory used by the trie structure mainly depends on the depth of the trie maintained (l), rather than on the cardinality (n) of the datasets. $\texttt{S}^3\texttt{J}$ uses more memory than $\texttt{S}^2\texttt{J}$, because $\texttt{S}^3\texttt{J}$ maintains a trie for both the outer and inner tables. Also, while the memory usage increases quickly with the depth of the trie, as the experiments in this section show, the overall depth that needs to be maintained is often not very high and the overall memory usage for the trie remains negligible.

Sorting and Z-order based B-tree Indexing Costs

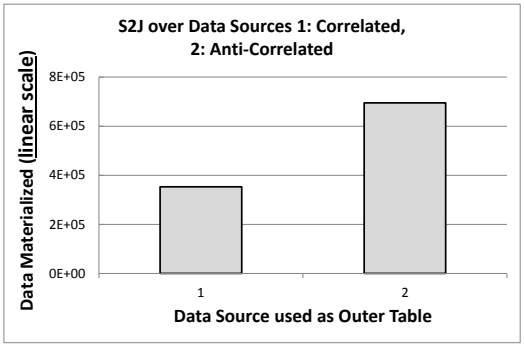
Table 3.6 shows that the time taken to preprocess the data before query execution is not very high. As expected, the costs increase with the increase in data cardinality, but this is negligible relative to the skyline-join query execution time; but comparing these with the execution times shown in Figure 3.20, we see that data preparation costs are effectively negligible relative to result enumeration times.

Effect of Data Correlation on Source Ordering

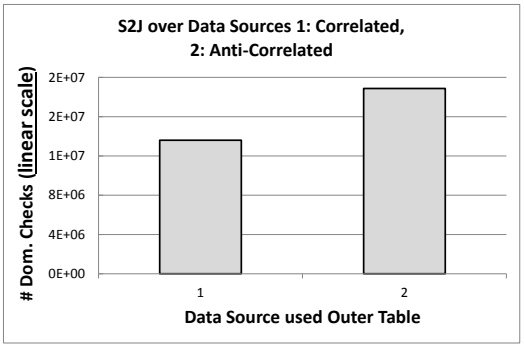
In this subsection, we study the effect of data correlation on the performance of $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$. In particular, we are interested in investigating whether the data correlation within a dataset gives us any insights into how to order the data sources in $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ processing. For this purpose, we consider scenarios in which the data sources



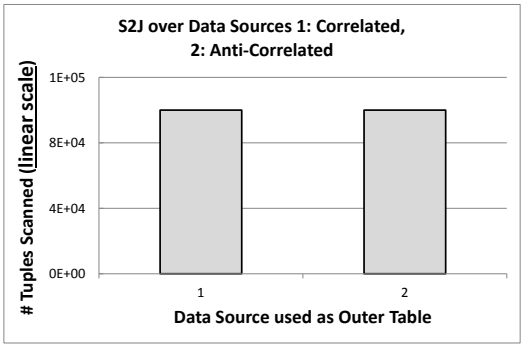
(a) Execution time



(b) Data materialized



(c) Dominance checks



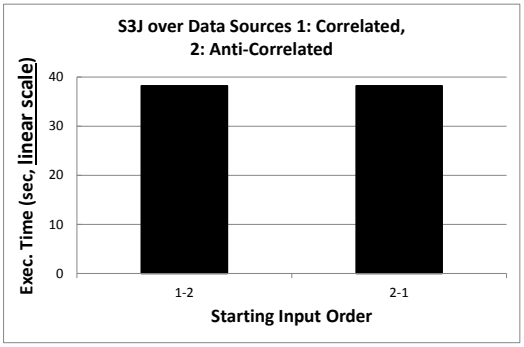
(d) Tuples scanned

Figure 3.22: Effect of data correlation on $\mathtt{S^2J}$; the x-axis presents strategies with different outer tables (Data source 1: Correlated, Data source 2: Anti-Correlated; $n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

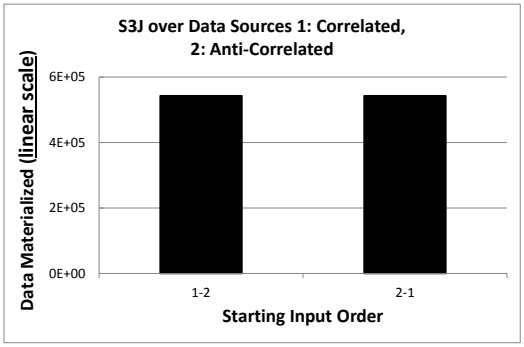
have different data distributions and see if this gives us any insights into finding a good order of the data sources for the $\mathtt{S^2J}$ and $\mathtt{S^3J}$ algorithms.

Figure 3.22 shows the performance of $\mathtt{S^2J}$ over non-identically distributed datasets in terms of execution time (Figure 3.22(a)), data materialized (Figure 3.22(b)), dominance checks (Figure 3.22(c)), and tuples scanned (Figure 3.22(d)). In these figures, the x-axis represents strategies with different outer tables; data source 1 is correlated, whereas data source 2 is anti-correlated. As can be seen in Figure 3.22(a),

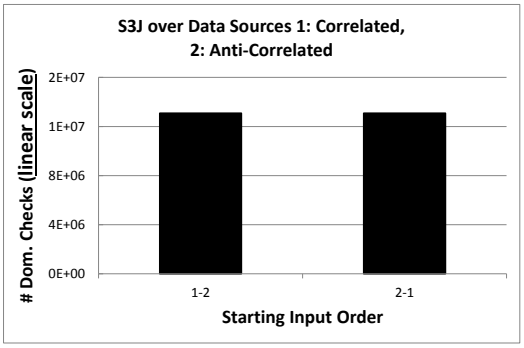
- *the execution time of* $\mathtt{S^2J}$ *is lowest when the data source used as the outer table is correlated (i.e., data source 1).*



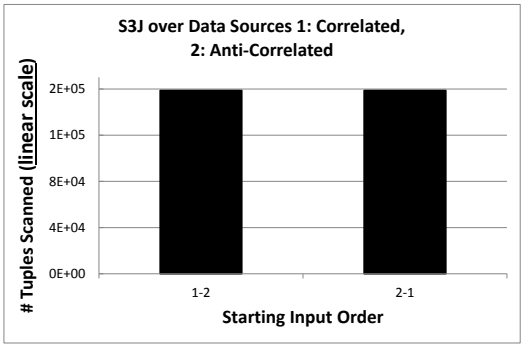
(a) Execution time



(b) Data materialized



(c) Dominance checks



(d) Tuples scanned

Figure 3.23: Data correlation does not impact the order in which data sources should be considered in $\mathtt{S^3J}$ (Data source 1: Correlated, Data source 2: Anti-Correlated; $n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

This is because when the correlated dataset (data source 1) is used as the outer table, $\mathtt{S^2J}$ materializes fewer number of candidates (Figure 3.22(b)) and performs a lower number of dominance checks (Figure 3.22(c)). Having a correlated data source as the outer table helps the LR-pruning strategy prune the inner table faster. This is because the bounds used by LR-pruning to mark the inner table (Section 3.3.2) are higher when the outer table has a correlated data distribution, thus the pruning obtained in this case is more.

Figure 3.23 shows the performance of $\mathtt{S^3J}$ over non-identically distributed data sources in terms of execution time (Figure 3.23(a)), data materialized (Figure 3.23(b)), dominance checks (Figure 3.23(c)), and tuples scanned (Figure 3.23(d)).

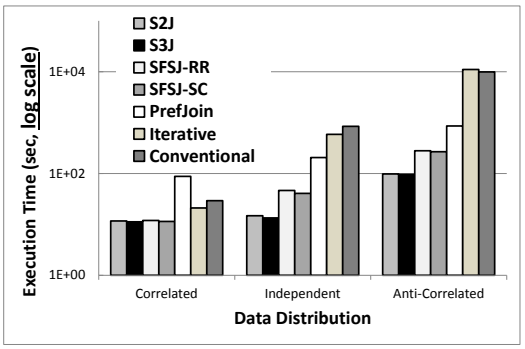


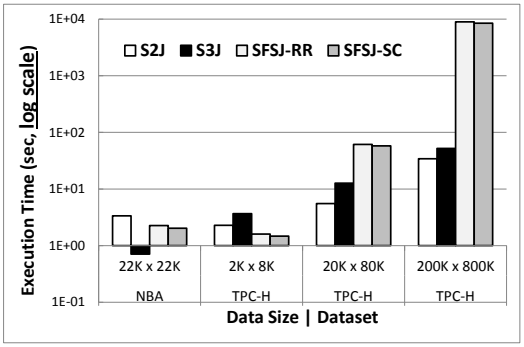
Figure 3.24: Comparison against other algorithms over correlated, independent and anti-correlated data sources ($n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

As before, data source 1 is correlated, whereas data source 2 is anti-correlated. However, since S^{3}J continuously swaps the roles of the outer and inner tables, the x-axis represents the initial order in which the two data sources are considered. As can be observed, the performance of S^{3}J is the same over all combinations of starting input orders. This means that even when the input data sources have a non-identical data distribution, the S^{3}J algorithm performs equally well regardless of which input order is chosen to start the skyline-join process. This implies that the swapping of the outer and inner tables in S^{3}J eliminates the need to pick one of the datasets as the "best" outer table even when the input data sources have different data distributions.

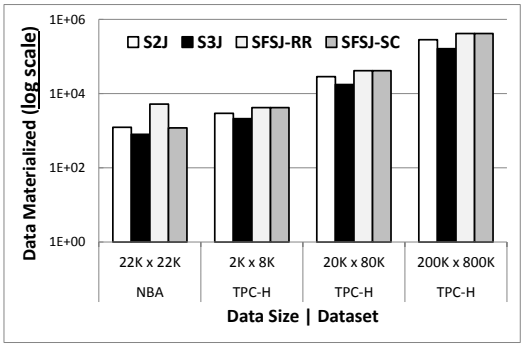
In the next section, we compare S^{2}J and S^{3}J against other skyline-join techniques and, in Section 3.5.3, provide detailed analysis of these algorithms under additional parameters, including data dimensionality, data cardinality, and join rate.

3.5.3 Evaluation of S^{2}J *and* S^{3}J *against other Techniques*

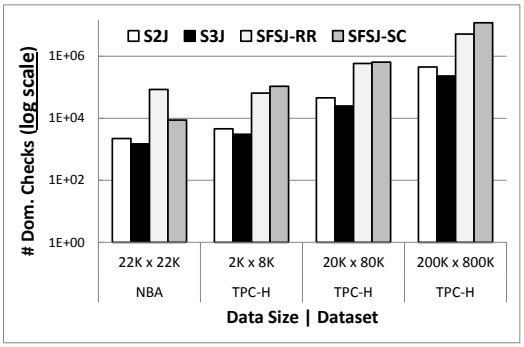
As we see in Figure 3.20, the S^{2}J and S^{3}J algorithms are most similar to each other in terms of execution time when we set $l = 5$. We have, therefore, set $l = 5$ as the maximum allowed depth of the trie for the experiments shown in the rest of the chapter. Note that, as shown in Figure 3.20, S^{3}J would be even faster if we use larger



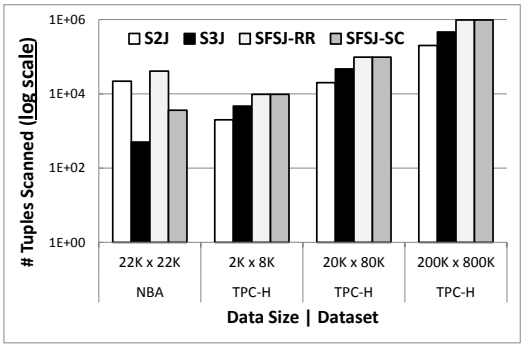
(a) Execution time



(b) Data materialized



(c) Dominance checks



(d) Tuples scanned

Figure 3.25: Performance on real (NBA) and benchmark (TPC-H) datasets ($j = 1$, $d = 2$, $s = 4$, $l = 5$)

values of l since finer resolution pruning helps in early stopping.

In Figure 3.24, we first compare $\mathtt{S^2J}$ and $\mathtt{S^3J}$ against the SFSJ methods [78], PrefJoin [36], Iterative skyline-join [68], and Conventional skyline-join (SFS version) for a sample scenario. As can be seen here, $\mathtt{S^2J}$, $\mathtt{S^3J}$ and the state-of-the-art approach, SFSJ, are the most competitive among the other alternatives, and the difference from the other techniques is very large. Therefore, in the rest of the section, we focus on the detailed study of our approaches and the SFSJ family of algorithms.

TPC-H Benchmark and NBA Dataset

We first use the TPC-H and NBA datasets to compare the S^{2}J and S^{3}J algorithms to SFSJ-RR and SFSJ-SC.

• **TPC-H Datasets.** TPC-H is a decision support benchmark. The TPC-H datasets were generated using the TPC-H DBGEN tool[10]. We ran the following query:

```
Skyline = SSJ * FROM Part P, PartSup PS,
                WHERE P.partkey = by PS.partkey,
                P.size MAX, P.retailPrice MAX,
                PS.availQty MAX, PS.supplyCost MAX
```

The cardinalities of the *Part* and *PartSup* tables were varied by choosing different *scale factor*s (SF). SF scales the database population – for a particular value of SF the DBGEN tool produces SF×200K tuples in table *Part* and SF×800K tuples in table *PartSup*.

Figure 3.25 illustrates that the S^{2}J and S^{3}J algorithms outperform SFSJ-RR and SFSJ-SC on the TPC-H datasets over all evaluation metrics. Please note that all plots are on a log scale. In terms of execution time (Figure 3.25(a)), both S^{2}J and S^{3}J show significant gains as the size of the datasets increases.

As is expected, on extremely small TPC-H datasets (2K x 8K), both S^{2}J and S^{3}J are marginally slower since they have the added overhead ($\sim$ 0.8 sec. for S^{2}J and $\sim$ 2 sec. for S^{2}J) of accessing the trie structure to find the regions that have been pruned. But this overhead becomes negligible as the size of the dataset grows; this is mainly because S^{2}J and S^{3}J leverage the block-based LR-pruning technique to materialize fewer number of intermediate skyline candidates (Figure 3.25(b)) and perform lesser number of dominance checks (Figure 3.25(c)) as compared to the SFSJ

[10]See Footnote 6.

Table 3.7: Percentage gain over TPC-H: 200K x 800K benchmark dataset ($j = 1$, $d = 2$, $s = 4$, $l = 5$)

(a) Execution Time

vs.	SFSJ-RR	SFSJ-SC
S²J	%99.6	%99.6
S³J	%99.4	%99.4

(b) Data materialized

vs.	SFSJ-RR	SFSJ-SC
S²J	%31.7	%31.7
S³J	%61.1	%61.1

(c) Dominance checks

vs.	SFSJ-RR	SFSJ-SC
S²J	%91.3	%96.2
S³J	%95.6	%98.1

(d) Tuples scanned

vs.	SFSJ-RR	SFSJ-SC
S²J	%79.5	%79.5
S³J	%52.4	%52.4

methods. Both SFSJ-RR and SFSJ-SC use time-consuming pairwise tuple-to-tuple dominance checks in order to eagerly prune tuples that cannot produce skyline-join results, hence this causes the cost of finding the skyline to be considerably higher than our algorithms. In addition, S³J has a tighter stopping condition as compared to SFSJ-RR and SFSJ-SC (Figure 3.25(d)). It accesses lesser number of tuples than the SFSJ methods, hence showing that the early stopping condition employed by S³J is successfully able to avoid more number of redundant accesses to the tuples in the input datasets.

An interesting observation is that under this parameter setting[11] the S²J algorithm performs slightly better than S³J on the TPC-H datasets in terms of execution time (Figure 3.25(a)), even though the S³J algorithm materializes fewer number of intermediate candidates (Figure 3.25(b)) and performs lesser number of dominance checks (Figure 3.25(c)) than S²J. This is because S³J scans more tuples as compared to S²J (Figure 3.25(d)), which only scans the tuples in the outer table. This results in additional scanning and pruning overheads in S³J that eliminate any gains

[11]Remember that the value of the parameter $l(= 5)$ is selected such that S²J and S³J perform similarly. Larger values of l would prune the space better and lead to further savings for S³J.

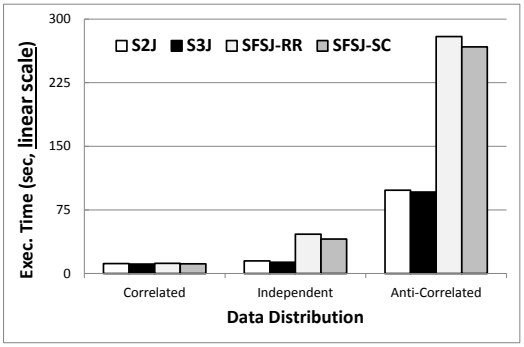
it achieves because of fewer candidates materialized or lesser number of dominance checks performed.

Finally, Table 3.7 summarizes the gains in terms of percentages over the TPC-H: 200K x 800K benchmark dataset. As can be seen, both $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ have significant gains over the SFSJ techniques.

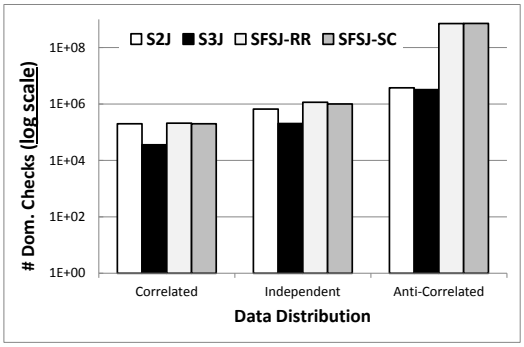
• **NBA Dataset.** For the experiments with the NBA dataset, we use the table that lists a player's regular season statistics – this table contains 21961 tuples and includes 17 types of statistics (e.g., assists and points) for the players. We encoded this dataset in the form of two tables: *Player-points (playerID, points, fieldGoals)* and *Player-assists (playerID, assists, freeThrows)*. The following query was run:

```
Skyline = SSJ * FROM Player-points P, Player-assists A,
                WHERE P.playerID = A.playerID,
                P.points MAX, P.fieldGoals MAX,
                A.assists MAX, A.freeThrows MAX
```

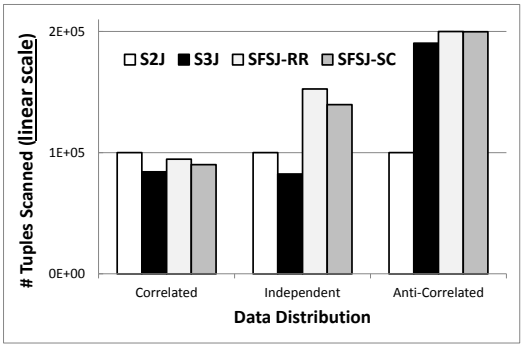
As Figure 3.25 shows, the trends are similar to the results obtained using the TPC-H datasets. $\texttt{S}^3\texttt{J}$ outperforms SFSJ on the NBA dataset as well. On the other hand, $\texttt{S}^2\texttt{J}$ is marginally slower since the added overhead ($\sim$ 1 sec.) of accessing the trie structure to find the regions that have been pruned is not overcome by the gains on this smaller dataset. Also, unlike the behavior observed on the TPC-H datasets, the $\texttt{S}^3\texttt{J}$ algorithm outperforms $\texttt{S}^2\texttt{J}$ on the NBA dataset in terms of execution time (Figure 3.25(a)). This is because $\texttt{S}^3\texttt{J}$ scans fewer number of tuples as compared to $\texttt{S}^2\texttt{J}$ (Figure 3.25(d)), thus adding to the gains $\texttt{S}^3\texttt{J}$ achieves by materializing fewer intermediate candidates (Figure 3.25(b)) and performing lesser number of dominance checks (Figure 3.25(c)).



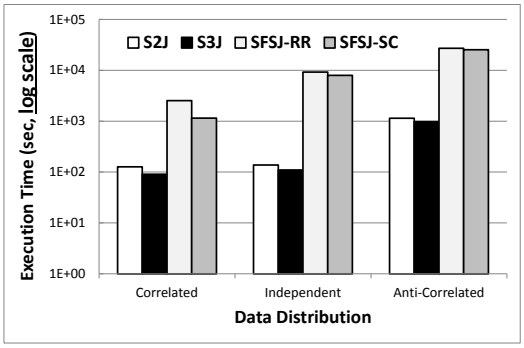
(a) Execution time for $n = 100K/dataset$



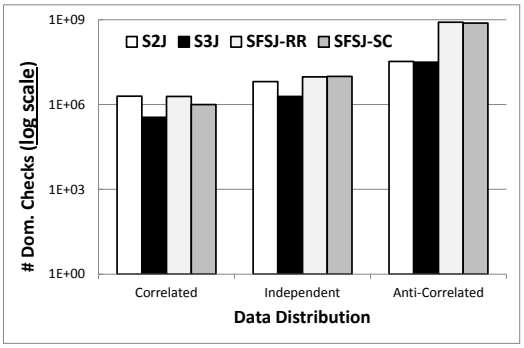
(b) Dominance checks for $n = 100K/dataset$



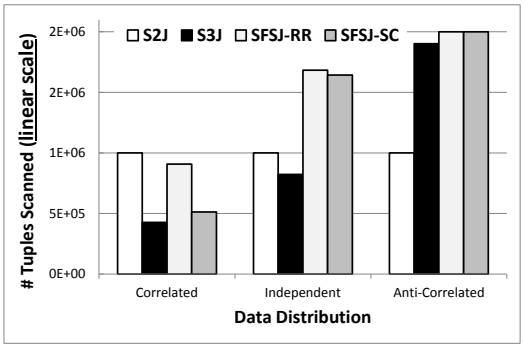
(c) Tuples scanned for $n = 100K/dataset$



(d) Execution time for $n = 1000K/dataset$



(e) Dom. checks for $n = 1000K/dataset$



(f) Tuples scanned for $n = 1000K/dataset$

Figure 3.26: Effect of data distribution over $n = 100K/dataset$ (Figures a, b, c) and $n = 1000K/dataset$ (Figures d, e, f) ($j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

Detailed Analysis using Synthetic Datasets

We now present a more detailed performance evaluation using synthetic datasets.

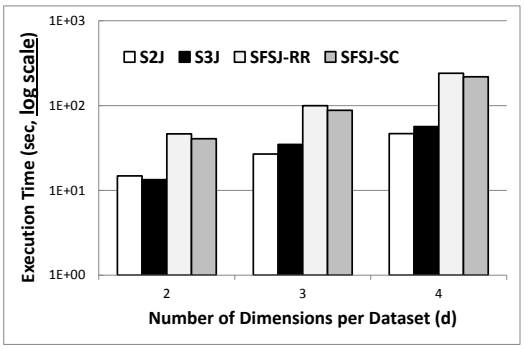
- **Effect of Data Distribution.** Figure 3.26 shows the performance of the al-

gorithms over data with different distributions in terms of execution time (Figures 3.26(a) and 3.26(d)), dominance checks (Figures 3.26(b) and 3.26(e)) and total number of tuples scanned (Figures 3.26(c) and 3.26(f)) over $n = 100K/dataset$ and $n = 1000K/dataset$. As illustrated in the figure, while $\mathtt{S^2J}$ and $\mathtt{S^3J}$ have only marginal gain on correlated datasets when $n = 100K/dataset$ (in which only a few skyline points are obtained while the majority of the data points are dominated), they perform extremely well on the larger correlated datasets ($n = 1000K/dataset$). Also, $\mathtt{S^2J}$ and $\mathtt{S^3J}$ perform very well on independent and anti-correlated data distributions at both $n = 100K/dataset$ and $n = 1000K/dataset$.

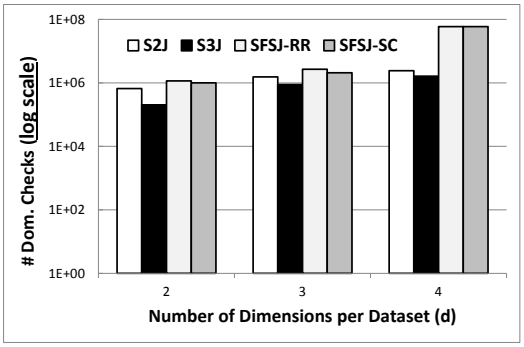
These show that the LR-pruning technique used by our algorithms and the early stopping condition used in $\mathtt{S^3J}$ are more effective than the corresponding pruning and stopping conditions in SFSJ, hence resulting in fewer number of dominance checks (Figures 3.26(b) and 3.26(e)) and lesser scans (Figures 3.26(c) and 3.26(f)). It is interesting to note that, on anti-correlated datasets, $\mathtt{S^2J}$ (counter-intuitively) scans less tuples than $\mathtt{S^3J}$. This is because, on anti-correlated data, the early stopping condition does not kick-in sufficiently early and due to the swapping of the roles of the input tables, more tuples are scanned overall. But due to the more effective pruning on both tables, $\mathtt{S^3J}$ results in a drop in dominance checks and as a result, $\mathtt{S^3J}$ leads to a better overall execution time.

In the rest of this subsection, we use datasets with independent data distribution.

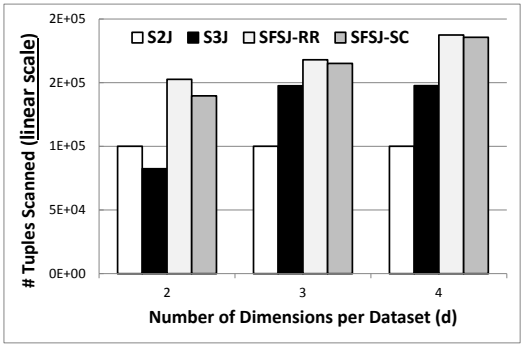
• **Effect of Dimensionality.** Figure 3.27 shows the impact of the number of skyline attributes per dataset ($d = 2, 3, 4$) across the various evaluation metrics. The total number of skyline attributes in each case is given by $s = 2 \times d$; i.e. 4, 6, and 8, respectively. As observed in the figures, the proposed algorithms outperform both SFSJ-RR and SFSJ-SC in terms of execution time (Figure 3.27(a)), dominance checks (Figure 3.27(b)), and total number of tuples accessed (Figure 3.27(c)). The SFSJ



(a) Execution time



(b) Dominance checks



(c) Tuples scanned

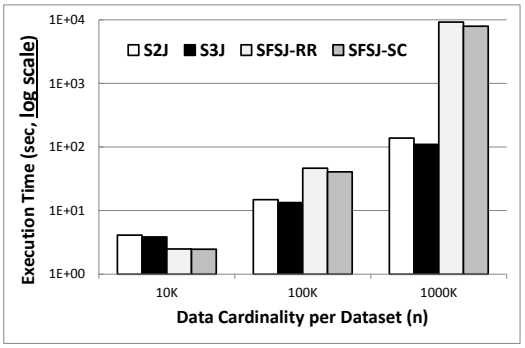
Figure 3.27: Effect of dimensionality over datasets with independent data distribution ($n = 100K/dataset$, $j = 1$, $s = 2d$, $r = 10$, $l = 5$)

methods fail to prune the space efficiently and incur additional cost due to its pruning process. This experiment illustrates that the proposed algorithms are more scalable as compared to SFSJ-RR and SFSJ-SC, and the trie-based LR-pruning technique is effective even on datasets with high dimensions.

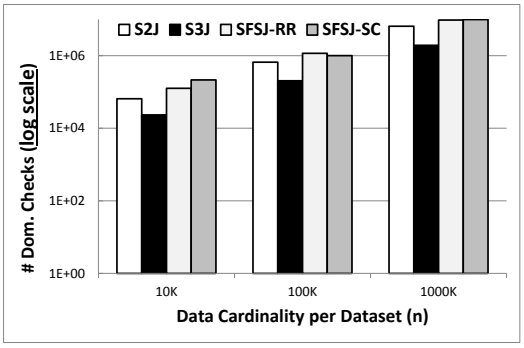
An important observation is that, when the number of skyline attributes per data source is larger than 2, the early-stopping condition of $\mathtt{S^3J}$ starts being less effective and, as a result, $\mathtt{S^3J}$ scans more tuples in total than $\mathtt{S^2J}$. Consequently, despite the reduction in the dominance checks due to more effective pruning of $\mathtt{S^3J}$, $\mathtt{S^2J}$ which scans less tuples, executes faster than $\mathtt{S^3J}$. Similar results were obtained for other data distributions.

Table 3.8: Impact of size of the outer table on execution time (Independent data sources, $j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

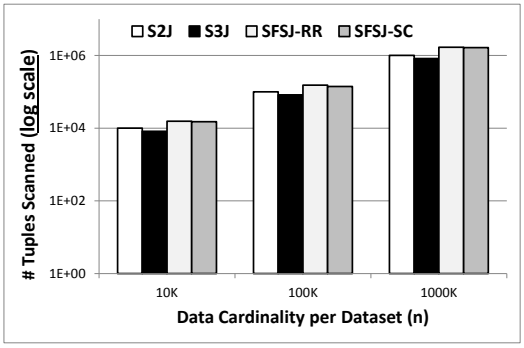
Outer Table Size	10K	100K	1000K
S^{2}J	4.11 sec.	14.82 sec.	137.87 sec.
S^{3}J	3.84 sec.	13.36 sec.	109.78 sec.
SFSJ-RR	2.49 sec.	46.31 sec.	9208.12 sec.
SFSJ-SC	2.47 sec.	40.59 sec.	7932.35 sec.



(a) Execution time



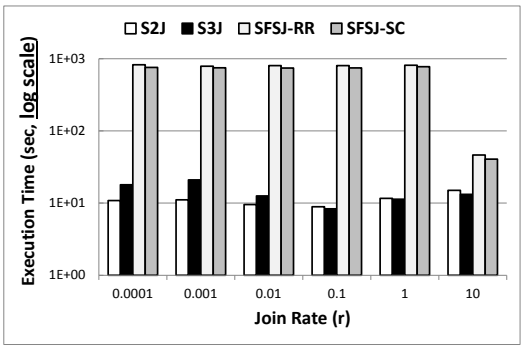
(b) Dominance checks



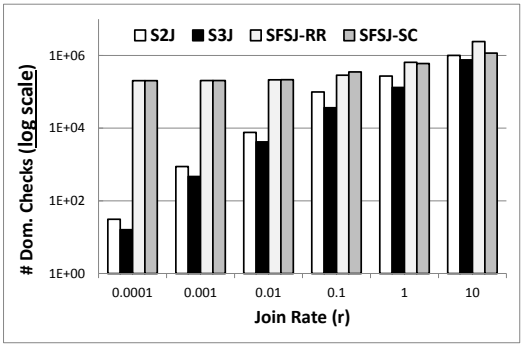
(c) Tuples scanned

Figure 3.28: Effect of data cardinality over independent data sources ($j = 1$, $d = 2$, $s = 4$, $r = 10$, $l = 5$)

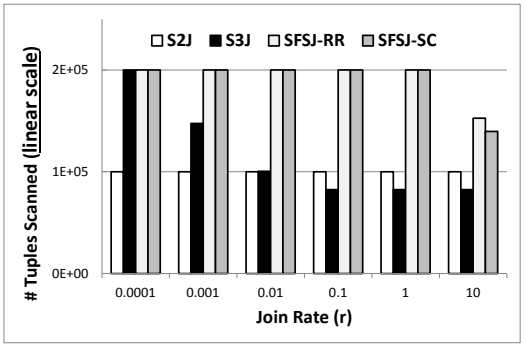
• **Effect of Data Cardinality.** S^{2}J and S^{3}J differ from each other in one key aspect. The S^{2}J algorithm makes a distinction between the outer and inner tables. Table 3.8 shows the impact of the size of the outer table on S^{2}J. As the size of the outer table increases the execution time increases as well. S^{3}J improves on S^{2}J by continuously



(a) Execution time



(b) Dominance checks



(c) Tuples scanned

Figure 3.29: Effect of join rate over independent data sources ($n = 100K/dataset$, $j = 1$, $d = 2$, $s = 4$, $l = 5$)

swapping the roles of the outer and inner tables for each layer. The swapping of the tables eliminates the need to pick one of the tables as the "best" outer table. This hypothesis is confirmed by the experimental results obtained. For instance, on tables containing 1 million tuples each (Table 3.8), the $\mathtt{S^2J}$ algorithm has an execution time of $\sim$ 137 sec., whereas $\mathtt{S^3J}$ has a faster execution time of $\sim$ 109 sec. Note that, $\mathtt{S^2J}$ still performs better than both SSFJ-RR and SFSJ-SC. On the above-mentioned dataset, the SSFJ-RR and SFSJ-SC methods have execution times of $\sim 9,208$ sec. and $\sim 7,932$ sec., respectively.

Figure 3.28 illustrates the performance of $\mathtt{S^2J}$ and $\mathtt{S^3J}$ against increases in data cardinality of each dataset (n = 10K, 100K, 1000K). The plots are on a log scale

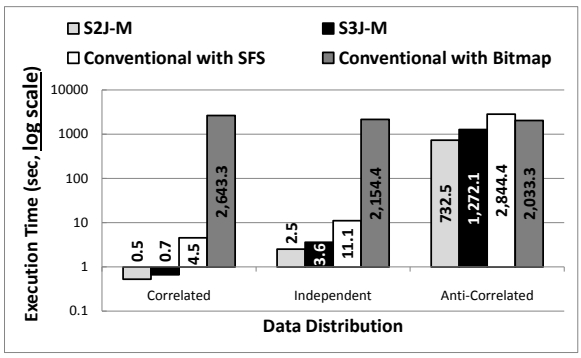
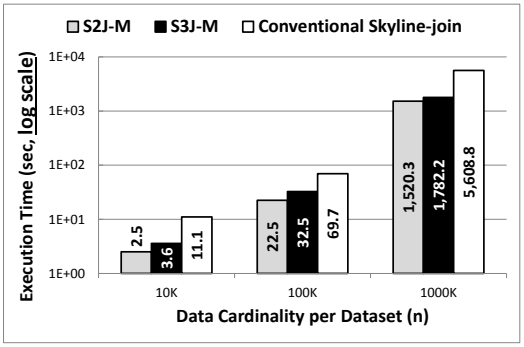


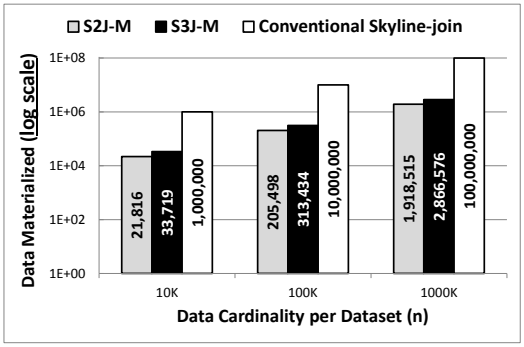
Figure 3.30: Comparison against conventional approach over correlated, independent and anti-correlated data sources ($j = 1$, $M = 3$, $n = 10K/dataset$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

and show results on independently distributed datasets. Figure 3.28(a) shows that both $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ scale well and perform significantly better in terms of execution time as compared to the SFSJ algorithms. On the very small dataset, $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ are marginally slower since they have the added overhead of accessing the trie structure in order to find the regions that have been pruned. But as the size of the datasets increases, much better than SFSJ-RR and SFSJ-SC because the LR-pruning technique helps our approaches prune the join space more effectively and prevent time-consuming tuple-to-tuple dominance checks whenever possible.

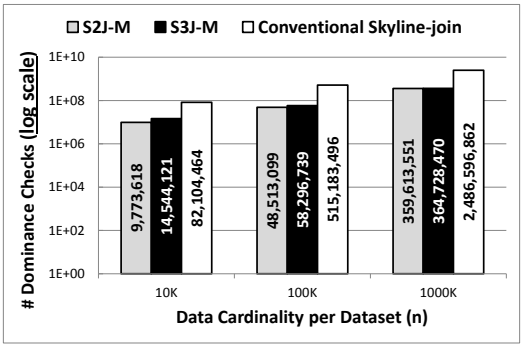
• **Effect of Join Rate.** Figure 3.29 compares the performance of $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ against the SFSJ methods across different join rates, namely $r = 0.0001, 0.001, 0.01, 0.1, 1, 10$. In terms of execution time (Figure 3.29(a)), the $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$ algorithms outperform both SFSJ-RR and SFSJ-SC across different joins rates. This gain is due to the fewer number of dominance checks (Figure 3.29(b)) performed by $\texttt{S}^2\texttt{J}$ and $\texttt{S}^3\texttt{J}$, and also the fact that they scan a lower number of tuples (Figure 3.29(c)) as they compute the skyline. This result shows that the proposed algorithms have a clear advantage over the SFSJ methods even when the join rates of the datasets are varied. This is because the LR-pruning technique is able to prune the join space more effectively than the SFSJ techniques, even in scenarios where the join rate is low.



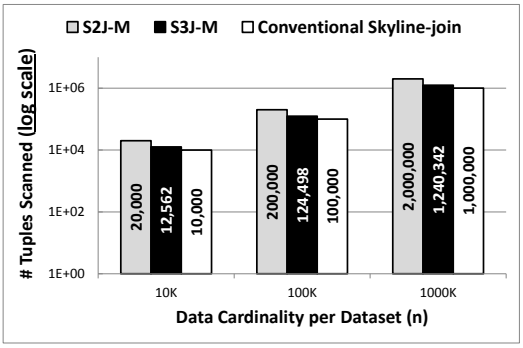
(a) Execution time



(b) Data materialized



(c) Dominance checks



(d) Tuples scanned

Figure 3.31: Comparison of S^{2}J-M and S^{3}J-M against conventional approach over independently distributed data sources of different cardinalites, n ($M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

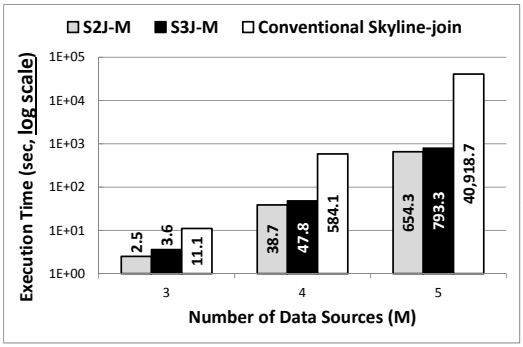
Another interesting observation that can be made from these results is that the S^{2}J algorithm performs better than S^{3}J in terms of execution time (Figure 3.29(a)) when the join rate is very low ($r = 0.0001, 0.001, 0.01$), even though the S^{3}J algorithm performs lesser number of dominance checks (Figure 3.29(b)) than S^{2}J. This is because S^{3}J scans more tuples as compared to S^{2}J (Figure 3.29(c)), which only scans the tuples in the outer table. This results in additional scanning and pruning overheads in S^{3}J that cancel out any gains it achieves because of lesser number of dominance checks performed. But at higher join rates ($r = 0.1, 1, 10$), the gains achieved by S^{3}J due to the pruning on both tables overcome the overheads incurred during pruning. Thus, in this case, S^{3}J leads to a better overall execution time.

3.5.4 *Evaluation of the* S^{2}J-M *and* S^{3}J-M *Algorithms*

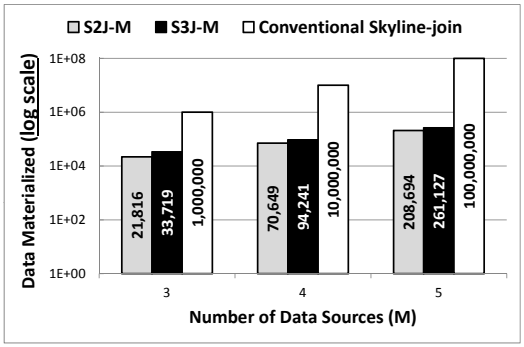
Since the SFSJ family of algorithms [78] are only applicable when processing two-way skyline-join queries, in this section, we first compare the S^{2}J-M and S^{3}J-M algorithms against the conventional skyline-join approach, in which we completely join the relevant data sources in order to materialize all candidate tuples, and then apply an existing single-source skyline algorithm [15] to obtain the skyline-join results. We also consider a bitmap-based alternative, which uses the *bitmap* skyline algorithm proposed in [70] to compute the skyline results. In our implementation of the *bitmap* skyline algorithm, the bitwise operations are carried out over compressed bitmaps. Since (as we show in Section 3.5.4) their execution time overheads are negligible and benefits are significant, in the experiments presented in this section, we use both OPT 1 and OPT 2 (based on the product monotone scoring function, $P(t)$) optimizations, discussed in Section 3.4.4, by default.

S^{2}J-M vs. S^{3}J-M vs. Conventional Skyline-Join vs. Bitmap Skyline-Join

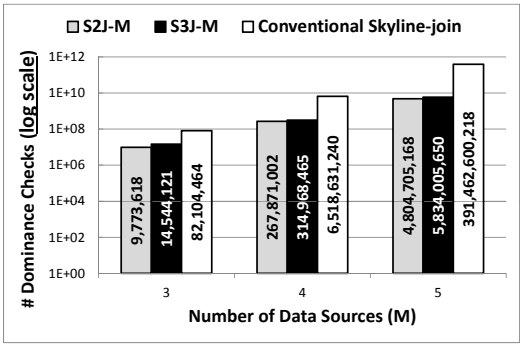
As can be seen in Figures 3.30, 3.31, and 3.32, S^{2}J-M and S^{3}J-M algorithms are significantly more efficient than alternative schemes under various data distributions, data sizes, number of data sources. Note that, as we see in Figure 3.30, except for when the data is anti-correlated, the bitmap-based approach is not competitive and has couple of orders slower execution time even for small datasets. Therefore, in Figures 3.31 and 3.32 and the rest of this section, we omit results corresponding to this approach. Note also that for the experiments included in Figure 3.32, we also ran configurations with six data sources, i.e., $M = 6$. However, for that case, the conventional skyline-join approach did not complete. Thus in Figure 3.32, we report the execution times only till $M = 5$. For $M = 6$, S^{2}J-M has an execution time



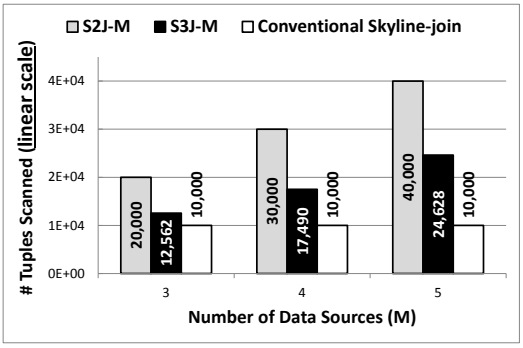
(a) Execution time



(b) Data materialized



(c) Dominance checks



(d) Tuples scanned

Figure 3.32: Comparison of S^{2}J-M and S^{3}J-M against conventional approach over different number of data sources, M, on independent data sources ($n = 10K/dataset$, $j = 1$, $d = 2$, $s = 2M$, $r = 10$, $l = 5$; note that the total number of skyline attributes is given by $s = 2 \times M$)

of $\sim 13,600$ sec., whereas S^{3}J-M completes query execution in $\sim 14,675$ sec. Also, as seen in Figures 3.31(d) and 3.32(d), conventional skyline-join scans lesser tuples than our approaches. The is because conventional skyline-join materializes candidate tuples by scanning only one of the multiple input datasets while it performs the join using random look-ups on the other tables. But, as shown by the results, the fact that conventional skyline-join materializes significantly more number of tuples and performs much higher number of dominance checks renders it inefficient.

Table 3.9: The following two sample configurations shows that S^{3}J-M has better worst-case behavior than S^{2}J-M when data sources are heterogeneous ($j = 1$, $M = 3$, $n = 10K/dataset$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

	Configuration #1 with three data sets 1: Anti-Corr., 2: Corr., 3: Anti-Corr.	Configuration #2 with three data sets 1: Anti-Corr., 2: Corr., 3: Corr.
S^{2}J-M	(worst-case) 41.84 sec.	(worst-case) 6.29 sec.
S^{3}J-M	(worst-case) 35.24 sec.	(worst-case) 3.15 sec.

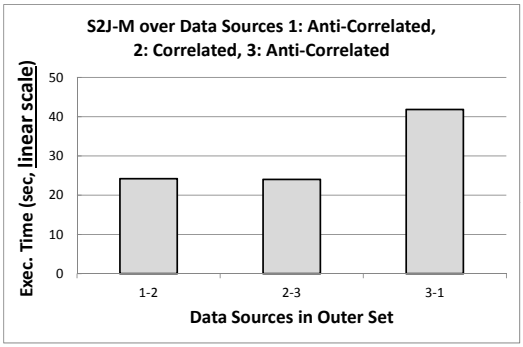
S^{2}J-M vs. S^{3}J-M

Note that these figures also show that, under the considered configuration, S^{2}J-M has a slightly better performance compared to S^{3}J-M. While this seems to indicate that it may be better to use the S^{2}J-M algorithm for processing skyline-join queries over more than two data sources, in general (as shown in Table 3.9) S^{3}J-M is more robust against differences in data distributions and data cardinalities and, consequently, avoids worst cases costs. This property of S^{3}J-M may render it more desirable in cases where advance knowledge of data is not available.

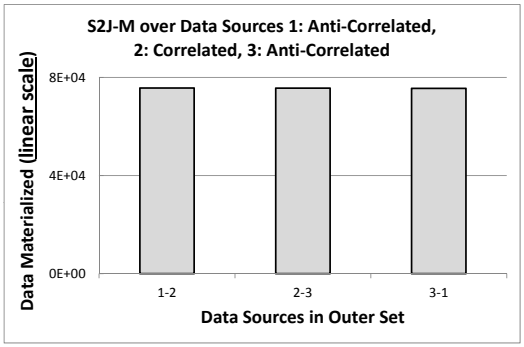
Therefore, in the following sections, we present an extensive experimental analysis of only the S^{2}J-M and S^{3}J-M algorithms. Based on these experimental results, we make recommendations on how to pick a good *skyline-sensitive join* query plan that may potentially lead to the most reduction in wasted work during the processing of skyline-join queries over more than two data sources.

Effect of Data Correlation on Source Ordering

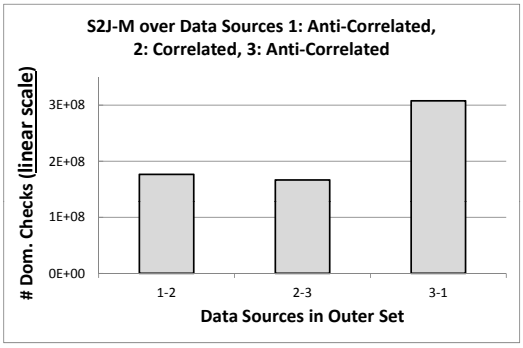
In this section, we study the effect of data correlation on the performance of S^{2}J-M and S^{3}J-M. In particular, we are interested in investigating whether the data correlation within a dataset gives us any insights as to how to order the data sources in S^{2}J-M and S^{3}J-M processing. For this purpose, we consider scenarios where three data



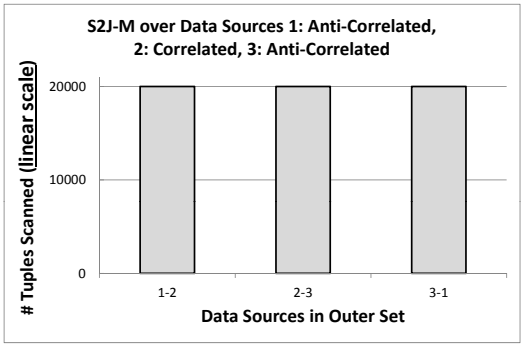
(a) Execution time



(b) Data materialized



(c) Dominance checks



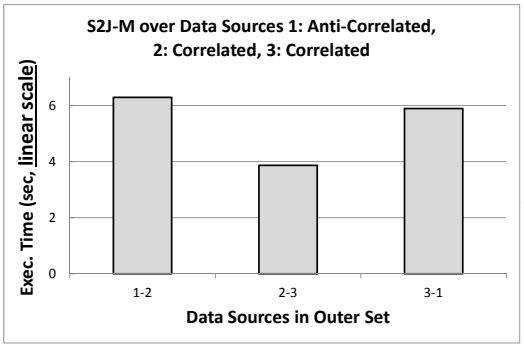
(d) Tuples scanned

Figure 3.33: Effect of data correlation on $\texttt{S}^2\texttt{J-M}$; the x-axis presents strategies with different outer sets (Data source 1: Anti-Correlated, Data source 2: Correlated, and Data source 3: Anti-Correlated; $M = 3$, $n = 10K/dataset$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

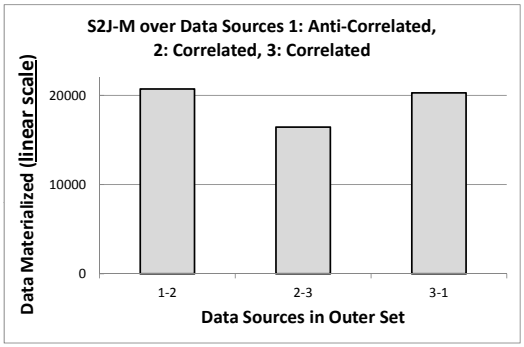
sources are differently distributed and see if this gives us any insights into finding good *skyline-sensitive join (*SSJ*)* query plans.

• **Impact of Data Distribution on the Processing of $\texttt{S}^2\texttt{J-M}$.** Figure 3.33 shows the performance of $\texttt{S}^2\texttt{J-M}$ over non-identically distributed data sources in terms of execution time (Figure 3.33(a)), data materialized (Figure 3.33(b)), dominance checks (Figure 3.33(c)), and tuples scanned (Figure 3.33(d)). In these figures, the x-axis represents strategies with different outer sets; datasets 1 and 3 are anti-correlated, whereas dataset 2 is correlated.

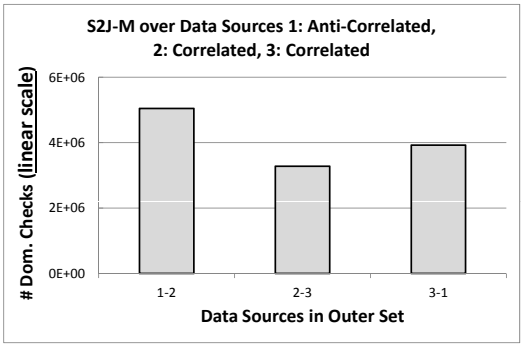
As can be seen in Figure 3.33(b), $\texttt{S}^2\texttt{J-M}$ materializes similar number of candidates



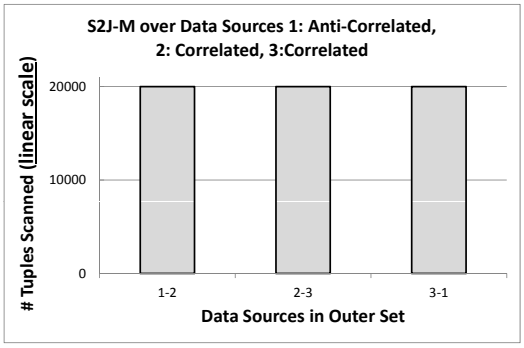
(a) Execution time



(b) Data materialized



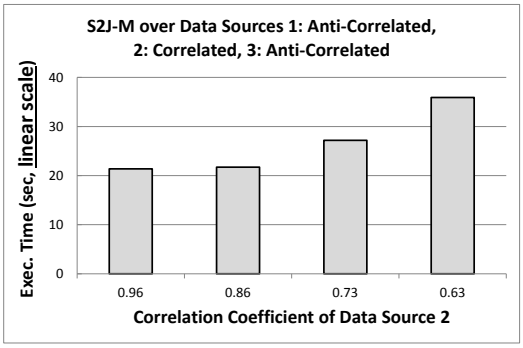
(c) Dominance checks



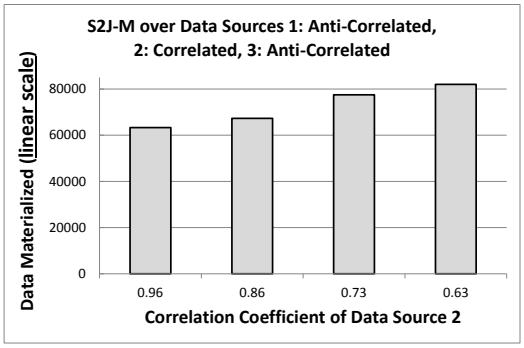
(d) Tuples scanned

Figure 3.34: Effect of data correlation on $\texttt{S}^2\texttt{J-M}$; the x-axis presents strategies with different outer sets (Data source 1: Anti-Correlated, Data source 2: Correlated, and Data source 3: Correlated; $M = 3$, $n = 10K/dataset$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

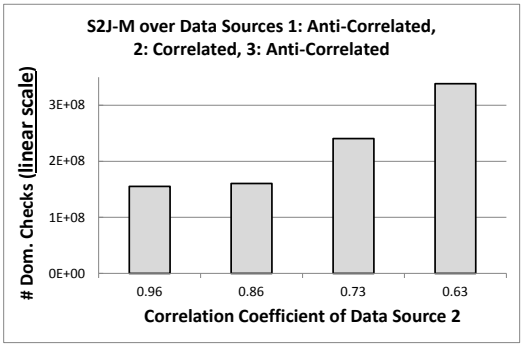
regardless of the contents of the outer set, but the execution time (Figure 3.33(a)) using SSJ query plans with data sources 1-2 and 2-3 in the outer sets is lower than the execution time using the query plan with data sources 3-1 in the outer set. This can be explained by the fact that query plans with 1-2 and 2-3 in the outer sets perform lower numbers of dominance checks (Figure 3.33(c)). Note that the outer set containing data sources 1-2 has one anti-correlated dataset (i.e., data source 1) and one correlated dataset (i.e., data source 2); this is also true in the case of the outer set containing data sources 2-3. In the case of the query plan with data sources 3-1 in the outer set, however, both data sources in the outer set are anti-correlated. This implies



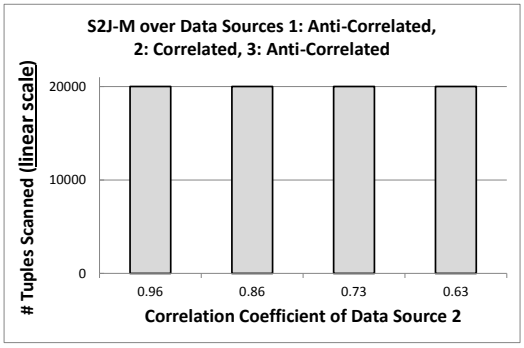
(a) Execution time



(b) Data materialized



(c) Dominance checks



(d) Tuples scanned

Figure 3.35: Effect of data correlation on $\texttt{S}^2\texttt{J-M}$; the x-axis represents the correlation coefficient of data source 2 (Data source 1: Anti-Correlated, Data source 2: Correlated, and Data source 3: Anti-Correlated; $M = 3$, $n = 10K/dataset$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

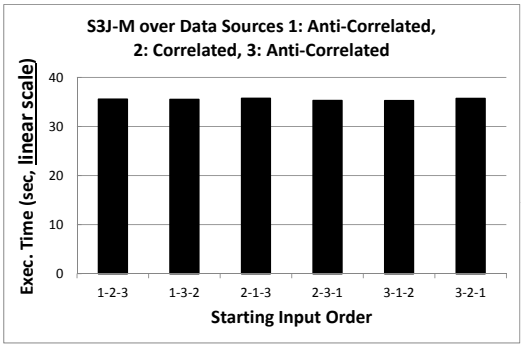
that query plans that contain at least one correlated data source in their outer sets are more suitable to skyline-join processing since these query plans materialize candidates in an order that is more conducive to skyline-join queries. Having correlated data sources in the outer set helps prune false-positives much faster and earlier, without having to scan the skyline-join candidate list very deep.

This result is confirmed in Figure 3.34, where datasets 2 and 3 are correlated, whereas dataset 1 is anti-correlated. As can be seen in this figure, *the execution time (Figure 3.34(a)) of* $\texttt{S}^2\texttt{J-M}$ *using* SSJ *query plans is lowest when both data sources (i.e., datasets 2 and 3) in the outer set are correlated.* Once again, as we see in

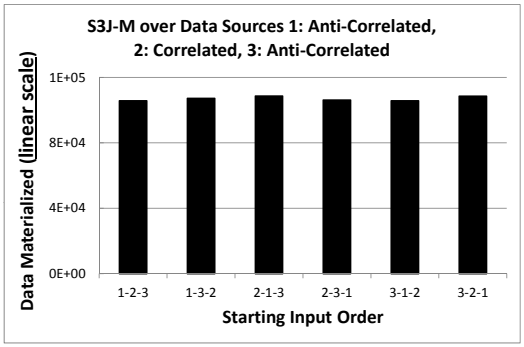
Figures 3.34(b) and 3.34(c), this is due the fact that the query plan with 2-3 in the outer set materializes fewer number of candidates and thus, performs a lower number of dominance checks. Having correlated data sources in the outer set helps the M-way LR-pruning strategy prune the inner table faster. This is because the bounds used by LR-pruning to mark the inner table (Section 3.4.2) are higher when the outer set contains only correlated data sources, thus the pruning obtained in this case is more.

Figure 3.35 further studies the scenario in which a query plan contains at least one correlated data source in its outer set, i.e. the query plan with data sources 1-2 in its outer set is used during skyline-join processing. Here, data sources 1 and 3 are anti-correlated, whereas dataset 2 is correlated. In this experiment, the correlation coefficient of data source 2 is varied from a high correlation coefficient of 0.96 to a low correlation coefficient of 0.63. As can be observed, the execution time (Figure 3.35(a)) of the $\texttt{S}^2\texttt{J-M}$ algorithm is the lowest when data source 2 is highly correlated and it increases as the correlation coefficient of data source 2 decreases. This is because $\texttt{S}^2\texttt{J-M}$ materializes larger number of candidates (Figure 3.35(b)) and thus, performs a higher number of dominance checks (Figure 3.35(c)) when the correlation coefficient of data source 2 decreases. This result implies that having highly correlated data sources in the outer set helps the M-way LR-pruning strategy prune the inner table faster. This is due to the fact that the bounds used by LR-pruning to mark the inner table are even more higher when the outer set contains a correlated data source with a high correlation coefficient.

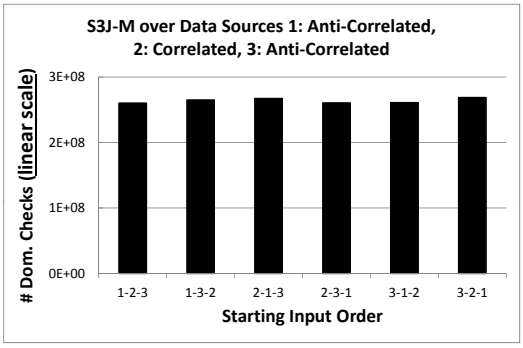
• **Impact of Data Distribution on the Processing of $\texttt{S}^3\texttt{J-M}$.** Figure 3.36 shows the performance of $\texttt{S}^3\texttt{J-M}$ over non-identically distributed data sources in terms of execution time (Figure 3.36(a)), data materialized (Figure 3.36(b)), dominance checks (Figure 3.36(c)), and tuples scanned (Figure 3.36(d)). As before, datasets 1 and 3 are anti-correlated, whereas dataset 2 is correlated. However, since in $\texttt{S}^3\texttt{J-M}$ the role



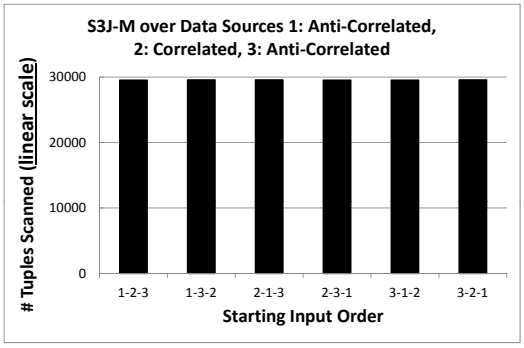
(a) Execution time



(b) Data materialized



(c) Dominance checks

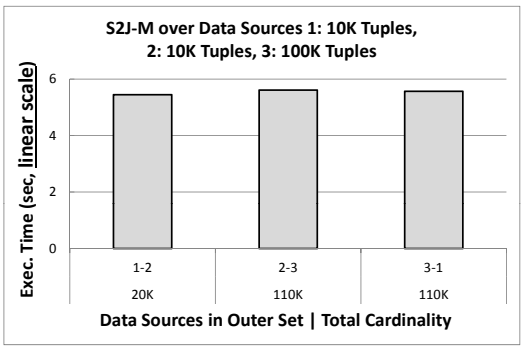


(d) Tuples scanned

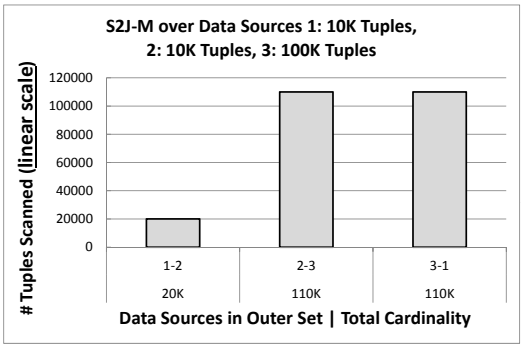
Figure 3.36: Data correlation does not impact the order in which data sources should be considered in $\texttt{S}^3\texttt{J-M}$ (Data source 1: Anti-Correlated, Data source 2: Correlated, and Data source 3: Anti-Correlated; $M = 3$, $n = 10K/dataset$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

of the outer set passes from one data source to the other at different iterations, the x-axis represents the initial order in which the different data sources are considered.

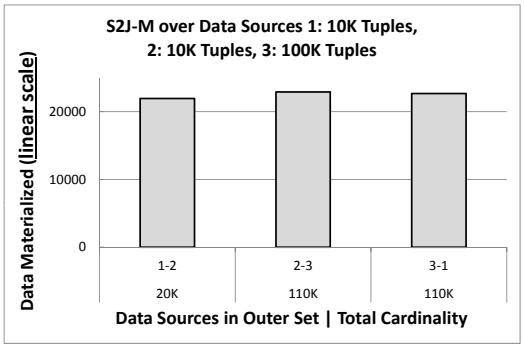
As can be observed, the performance of $\texttt{S}^3\texttt{J-M}$ is similar over all combinations of starting input orders and the difference in execution time (Figure 3.36(a)) is negligible. This means that even when the input data sources have a non-identical data distribution, the $\texttt{S}^3\texttt{J-M}$ algorithm performs equally well regardless of which input order is chosen to start the skyline-join process. This implies that the swapping of query plans in $\texttt{S}^3\texttt{J-M}$ eliminates the need to pick one of the combinations of outer sets as the "best" outer set even when the input data sources have different data distributions.



(a) Execution time



(b) Tuples scanned



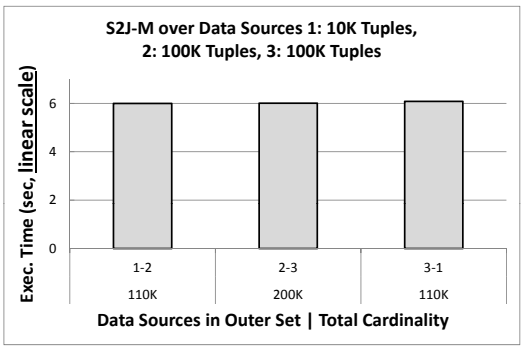
(c) Data materialized

Figure 3.37: Effect of data cardinality of outer set on S^{2}J-M over independent data sources 1: 10K Tuples, 2: 10K Tuples, and 3: 100K Tuples ($M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

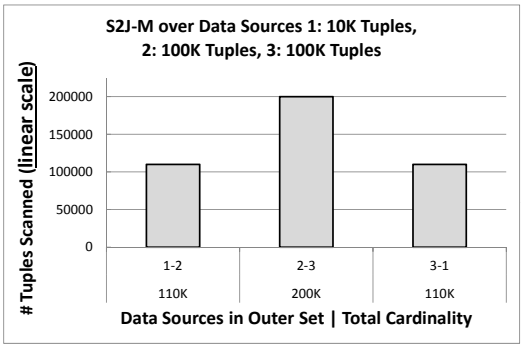
Effect of Non-Identical Data Cardinality on Query Plans

Here we investigate the effect of non-identical data cardinality of different data sources on the query plans for S^{2}J-M and S^{3}J-M. Note that the experiments in this section were carried out over independent data sources. Similar results were obtained for other data distributions.

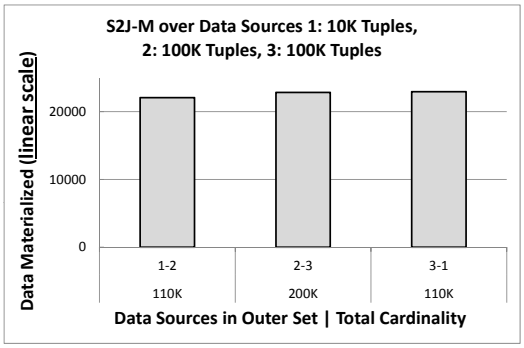
• **Impact of the Outer Set's Data Cardinality on S^{2}J-M.** Figures 3.37 and 3.38 show the effects of data cardinality of the outer set on S^{2}J-M. As it can be observed, the cardinality of the outer set affects the total number of tuples scanned (Figures 3.37(b), 3.38(b)), but only has a slight (on average $\sim$ 1.5%) effect on the



(a) Execution time



(b) Tuples scanned

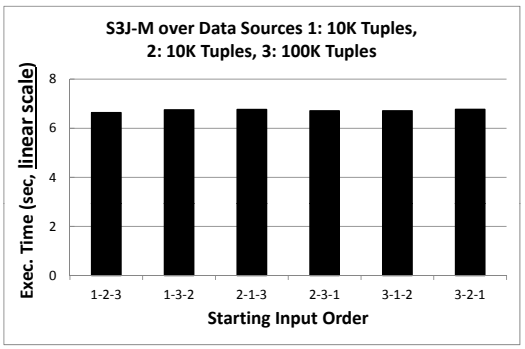


(c) Data materialized

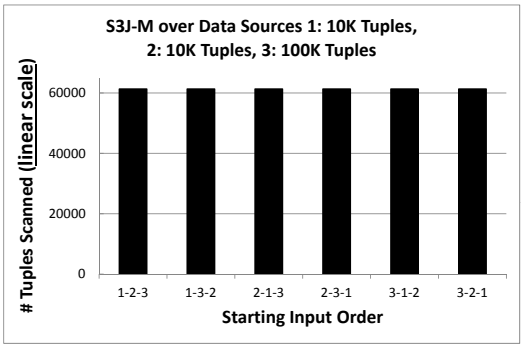
Figure 3.38: Effect of data cardinality of outer set on $\mathtt{S^2J\text{-}M}$ over independent data sources 1: 10K Tuples, 2: 100K Tuples, and 3: 100K Tuples ($M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

execution time (Figures 3.37(a), 3.38(a)). This is due to the fact that the execution time is primarily dependent on the number of intermediate data points materialized and the cardinality of the outer set does not significantly impact the number of intermediate results materialized (Figures 3.37(c), 3.38(c)). This is because the amount of data materialized dictates the extent of overhead on pruning the join space using LR-pruning. The larger the number of data points materialized, the higher is the time spent on maintaining the data structures that help in the pruning process.

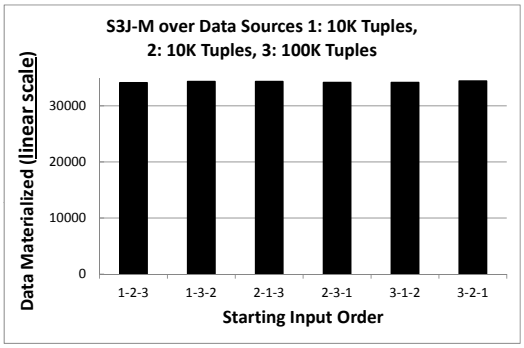
It is important, however, to recognize that the impact of the tuples scanned could have been higher in scenarios where the access to data from sources is costly. In those cases, we would recommend picking a `SSJ` query plan that has data sources



(a) Execution time



(b) Tuples scanned

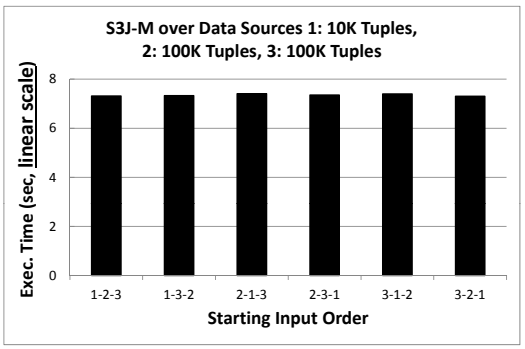


(c) Data materialized

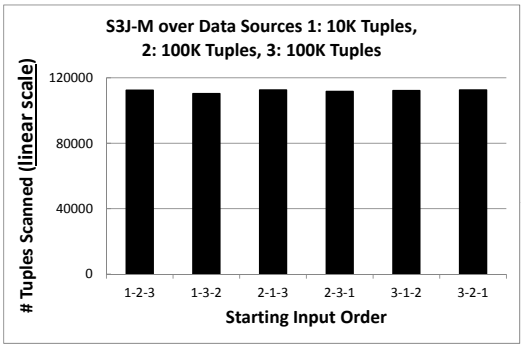
Figure 3.39: Non-identical data cardinality does not impact the order in which data sources should be considered in S^{3}J-M (Independent Data source 1: 10K Tuples, Data source 2: 10K Tuples, and Data source 3: 100K Tuples; $M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

with smaller cardinalities in its outer set, so that the amount of data scanned by the S^{2}J-M algorithm is minimized.

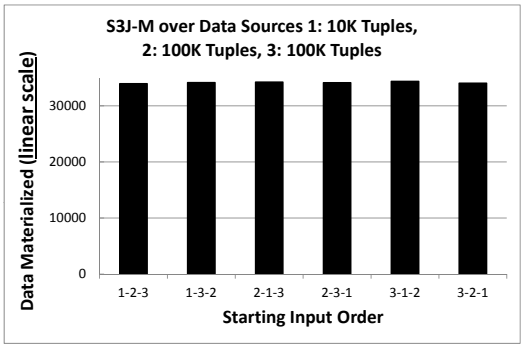
• **Impact of the Data Cardinality on the Processing of S^{3}J-M.** Figures 3.39 and 3.40 show the effect of non-identical data cardinality of inputs on S^{3}J-M in terms of execution time (Figures 3.39(a), 3.40(a)), tuples scanned (Figures 3.39(b), 3.40(b)) and data materialized (Figures 3.39(c), 3.40(c)). As it can be seen, the performance of S^{3}J-M is similar over all combinations of starting input orders and the difference in execution time is negligible. This means that even when the input data sources have non-identical data cardinalities, the S^{3}J-M algorithm performs equally well regardless



(a) Execution time



(b) Tuples scanned



(c) Data materialized

Figure 3.40: Non-identical data cardinality does not impact the order in which data sources should be considered in $\texttt{S}^3\texttt{J-M}$ (Independent Data source 1: 10K Tuples, Data source 2: 100K Tuples, and Data source 3: 100K Tuples; $M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

of which input order is chosen to start the skyline-join process. Thus, this provides evidence that the swapping of query plans in $\texttt{S}^3\texttt{J-M}$ eliminates the need to pick one of the combinations of outer sets as the "best" outer set even when the input data sources have different data cardinalities.

Impact of Using Additional Optimizations, OPT 1 and OPT 2

Finally, we provide a detailed discussion of the $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$ algorithms, we first investigate the effectiveness of the optimization techniques, namely OPT 1 and OPT 2, discussed in Section 3.4.4 to help us identify the default configuration to be used in

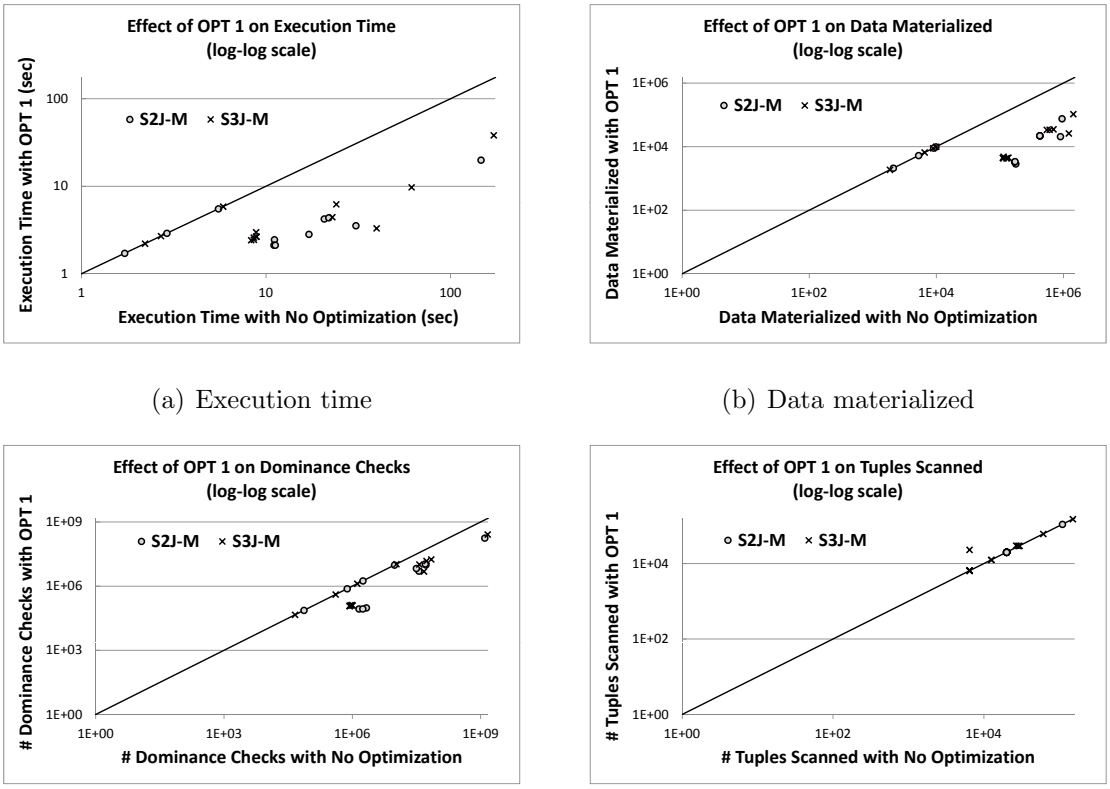
Table 3.10: Time overheads of OPT 1 and OPT 2 on a single data source (Independent data source, $d = 2$, $r = 10$)

Data Cardinality	10K	100K	1000K
OPT 1	56 msec.	390 msec.	4,936 msec.
OPT 2	39 msec.	236 msec.	1,655 msec.

the detailed evaluations. The data distributions of the datasets used in this section include correlated, independent and anti-correlated distributions.

• **Time Overheads of OPT 1 and OPT 2.** Before we study their benefits, we investigate the time overheads of implementing OPT 1 and OPT 2 optimizations. As can be observed from the results shown in Table 3.10, the time costs of the optimization techniques are negligible relative to the time cost of the skyline-join operations themselves.

• **Effect of OPT 1.** Figure 3.41 shows the performance of the OPT 1-optimized and non-optimized versions of `S²J-M` and `S³J-M` over data with various distributions, join rates, and cardinalities in terms of execution time (Figure 3.41(a)), number of intermediate data points materialized (Figure 3.41(b)), dominance checks (Figure 3.41(c)) and total number of tuples scanned (Figure 3.41(d)). As illustrated in this figure, OPT 1 helps reduce query execution time significantly as compared to the non-optimized versions of `S²J-M` and `S³J-M`. This is indicated by the fact that most of the plotted points fall below the 45° line (Figure 3.41(a)). As we see in Figures 3.41(b) and 3.41(c), the gains in execution time are primarily due to the fact that OPT 1 causes `S²J-M` and `S³J-M` to materialize lower number of candidates and perform fewer number of dominance checks. This is because OPT 1-optimized versions of `S²J-M` and `S³J-M` do not process any tuples that are not in the group skyline, hence preventing the generation of wasteful candidates that are guaranteed to not be a part of the final skyline-join result (except in cases where all the join attribute



(a) Execution time

(b) Data materialized

(c) Dominance checks

(d) Tuples scanned

Figure 3.41: Effect of OPT 1 on $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$ over correlated, independent and anti-correlated data sources ($M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

values of the datasets participating in the skyline-join are unique – i.e., $r = 1$ for all datasets in the skyline-join).

• **Effect of Using OPT 1 and OPT 2 Together.** Figure 3.42 illustrates the effect of both OPT 1 and OPT 2 on $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$ using the product monotone scoring function, $P(t)$, (results for the sum monotone function are similar) versus the OPT 1-optimized versions of $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$. As we see in this figure, in almost all cases, OPT 2 provides additional gains (on average $\sim$ 15%) in execution times on top of OPT 1, and is able to improve the efficiency of $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$ even in cases where they are otherwise on the the 45° line (Figure 3.42). The reason for this is that OPT 2, unlike OPT 1, is effective even when all the join attribute

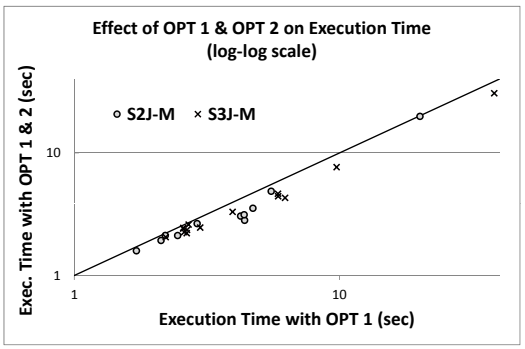


Figure 3.42: Effect of OPT 1 and OPT 2, instead of only OPT 1, on the execution time of $\texttt{S}^2\texttt{J-M}$ and $\texttt{S}^3\texttt{J-M}$ over correlated, independent and anti-correlated data sources ($M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

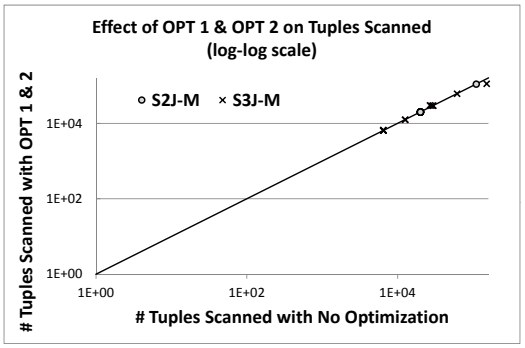


Figure 3.43: When using OPT 1 and OPT 2, the outlier cases with large number of scans are avoided; note that the results in this figure do not contain the outlier case in Figure 3.41(d) (Correlated, Independent and Anti-correlated data sources, $M = 3$, $j = 1$, $d = 2$, $s = 6$, $r = 10$, $l = 5$)

values of the datasets participating in the skyline-join are unique. As a result the number of dominance checks, on average, drops $\sim 1.5\%$ in the case of $\texttt{S}^2\texttt{J-M}$ and $\sim 2.8\%$ in the case of $\texttt{S}^3\texttt{J-M}$. It is important to note that, when all things (such as the number of intermediate join results materialized) are equal, the execution time is determined by the number of dominance checks and a small drop in the number of dominance checks (on the order of few percentage points) translates to significant gains (on avarage $\sim 15\%$) in execution time. Note that OPT 2, used hand-in-hand with OPT 1, also helps avoid the (rare) occurrences of $\texttt{S}^3\texttt{J-M}$ scanning deeper than its non-optimized version (compare Figures 3.41(d) and 3.43).

Chapter 4

LAYERED PROCESSING OF SKYLINE-WINDOW-JOIN QUERIES ON DATA STREAMS

4.1 Introduction

Given a set, D, of data points in a feature space, the *skyline* of D consists of the points that are not dominated[1] by any other data point in D [13]. Intuitively, the skyline is a set of *interesting* points that help paint the "bigger picture" of the data in question, providing insight into the diversity of the data across different data features. Searching for non-dominated data is valuable in many applications that involve multi-criteria decision making [65]. For example, a stock investor might find the skyline of stock market transactions useful in making trade decisions. The number of shares (volume) and price per share are two attributes that are typically used to characterize stock transactions. In Figure 4.1, the points that are connected represent stock transactions that are part of the skyline; this includes transactions that are low-priced or/and have a large trade volume at a given point in time. The skyline in this example represents transactions that are more *interesting* than the rest of the transactions with respect to one or both criteria. Other transactions are not in the skyline because they are dominated in terms of price per share and/or volume by at least one transaction that is in the skyline. The shaded area in Figure 4.1 is the dominance region of stock transaction b: for any transaction in the region shown, b is either cheaper (per share) and/or has a higher volume; therefore b can be said to

[1] A point dominates another point if it is as good or better in all dimensions, and better in at least one dimension.

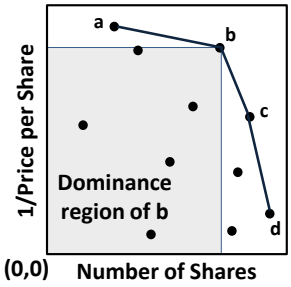


Figure 4.1: Skyline of stock transactions

be more *interesting* than all transactions it dominates.

Due to its ability to summarize large data sets into small but *interesting* subsets, efficient processing of skyline queries has received considerable attention. Most early works focused on cases where the data is static [13, 68]. However, the advent of data stream applications has motivated the development of techniques to address the unique requirements of skyline query processing over data streams, where rather than computing a single skyline, the system continuously tracks skyline changes in an incremental manner. Several algorithms [69, 19, 44, 71, 87] that consider the special characteristics of streams, such as fast data updates and strict limits for response time, have been proposed. These techniques support on-line computation of skylines over rapid data streams and are able to efficiently monitor skyline changes as tuples arrive/expire continuously.

4.1.1 Skylines over Multiple Data Streams

Existing data stream skyline algorithms, nevertheless, focus on single-stream skyline processing in which all required skyline attributes are present in the same stream. However, there are many applications that require integration of the data from different sources (e.g., data originating from different sensors or from different sources in a distributed publish/subscribe architecture) and, in such scenarios, the data stream skyline query may involve attributes belonging to different data streams, making the

join operation an integral part of the overall process. For instance, going back to our earlier example (Figure 4.1), a stock investor may also consider the risk of investing in a certain stock and the corresponding commission cost to be additional factors in his/her decision-making process. If this information is available as part of a second data stream, then the two streams would need to be "joined" before the skyline of stock transactions with respect to the price, volume, risk and commission cost can be found. Other join-based skyline applications can involve tracking objects through a network of sensors, or recording changes in large (but slow evolving) web-based data sets. Examples include generating click information about the number of visits to multiple web sites, or even monitoring the progress of cars through highway toll-booths based on specific attributes.

While the need to support skyline join queries over data streams is increasingly being recognized by the community [14], as of today, there are no algorithms that integrate skyline computation as a part of window-based join processing. As discussed earlier, existing techniques mainly focus on skyline processing over single-streams. These techniques can be used in conjunction with stream-join algorithms (by incrementally maintaining the join data using stream-join algorithms and then searching for skylines over the joined data stream), but in this work we note that this approach will introduce significant amounts of waste as, it has been shown for static data [68, 52], performing a skyline search *after* a join operation is almost always less efficient than integrating the skyline search with the join processing.

4.1.2 Contributions of this Work

This work studies skyline-window-join (SWJ) query processing over multiple data streams[2]. We propose a Layered Skyline-window-Join (LSJ) operator that maintains

[2]This work has been published in ICDE 2013.

skyline-join results in a layered, incremental manner by continuously monitoring the changes in the data streams, and eliminates redundant work between consecutive windows by leveraging shared skyline objects across all iteration layers of skyline-join processing. Our contributions are as follows:

- Firstly, we formally define the *skyline-window-join* (SWJ) operation over multiple data streams.

- Next, we develop a framework called the *iteration-fabric* that forms the backbone of the proposed layered, incremental skyline-window-join algorithms over data streams.

- The *iteration-fabric* helps us combine the advantages of two existing skyline methods, *StabSky* [44] and *Iterative* [68], in developing a `L`ayered `S`kyline-window-`J`oin algorithm (`LSJ`) that maintains skyline-join results in an incremental manner by continuously monitoring the changes in the input streams and leveraging any overlaps that exist between the data considered at individual processing layers of consecutive sliding windows.

- We test the efficiency and performance of the proposed algorithms. Extensive experimental evaluations over real and simulated data show that `LSJ` provides significant gains, especially on data with correlated skyline attributes, over alternative schemes which are not designed to eliminate redundant work across skyline-join layers, especially in scenarios with large data volumes and with considerable overlaps between consecutive windows.

To the best of our knowledge, this is the first research work to address join-based skyline queries over pairs of data streams. The rest of the chapter is structured as follows: Section 4.2 provides an introduction to join-based skyline queries over

Table 4.1: Notations used in this chapter

Notation	Description
S_i	stream $i \in \{1, 2\}$
$\mathcal{A}$	set of data attributes
$p.A$	value of attribute A of tuple p
Θ	join condition
$\mathcal{A}_\Theta$	set of join attributes
$\hat{\mathcal{A}}$	set of skyline attributes
$p.start$	generation timestamp of tuple p
$p.end$	expiration timestamp of tuple p
ω	size of a sliding window
σ	sliding window shift length
W_j	sliding window j
$W_{i,j}$	data for stream $i \in \{1, 2\}$ in sliding window j
$W_{i,l,j}$	data for stream $i \in \{1, 2\}$ at layer l in window j
$W^+_{i,l,j}$	set of tuples added at layer l for stream $i \in \{1, 2\}$ in window j
$W^-_{i,l,j}$	tuple set removed at layer l for stream $i \in \{1, 2\}$ in window j
$\Lambda_{i,l,j}$	local skyline set of layer l for stream $i \in \{1, 2\}$ in window j
$\mathcal{N}_{i,l,j}$	non-redundant tuple set in layer l, window j, stream $i \in \{1, 2\}$
$\mathcal{T}_{i,l,j}$	tree-structured dominance graph built on $\mathcal{N}_{i,l,j}$
$\mathcal{G}_{l,j}$	global skyline set of layer l, window j
$\mathcal{G}_j$	global skyline set of window j

sliding windows. In Section 4.3, we discuss of a naive implementation of the SWJ operation, which highlights the underlying challenges and points out to the opportunities we leverage when developing the novel algorithms proposed in this chapter. In Section 4.4, we discuss the proposed algorithms in detail. Section 4.5 presents an extensive experimental evaluation of the proposed approaches.

4.2 Preliminaries

In this section, we formally introduce the *skyline-window-join* (SWJ) model of processing continuous, sliding window skyline-join queries over data streams. Table 4.1 summarizes the notations used throughout this chapter.

4.2.1 Sliding Windows over Data Streams

This work focuses on skyline-joins over pairs of data streams bound by time-based sliding windows.

Sliding windows allow unbounded data streams to be limited to a certain size and finite number of states. As described in [5], the size of a window can be stated in terms of "logical" or "physical" units. *Time-based* sliding windows fall under the category of windows described using logical units. Each tuple, p, is *alive* in the system for a specific length of time; this is termed as the tuple's *lifespan* and is equal to $[p.start, p.end)$. Here, $p.start$ is the tuple's generation timestamp – the time at which the tuple was generated at the remote source. The expiry time, $p.end$, is the time when the tuple is removed from the window – this occurs when $p.start$ is no longer covered by the sliding window. The length/size of the sliding window is described by a parameter ω: each window W_j spans a time period from $[W_j.start, W_j.end)$, where $W_j.end = W_j.start + \omega$.

In the *count-based* sliding window model, on the other hand, a tuple expires after ω subsequent tuples have been received, regardless of their generation timestamps. In other words, in this model, only the tuples with ω most recent timestamps are considered. A second parameter used in defining sliding windows over data streams is the "shift" or σ: In time-based sliding windows, for example, the shift constraints the start of the window to shift σ units; i.e., $\forall_j W_{j+1}.start = W_j.start + \sigma$. In count-

based sliding windows, the shift defines how many tuples are skipped from one window to the next.

Note that count-based sliding windows can be treated as time-based windows by associating each tuple p with an artificial generation time, $p.start$, that equals its relative position in the stream (i.e., the first tuple arrived has position 1, the second 2, and so on). Therefore, in the rest of the chapter, without loss of generality, we assume that the sliding windows are *time-based*, as opposed to *count-based*.

4.2.2 Continuous Joins over Sliding Windows

Given (a) two streams, $S_1(A_1, \ldots, A_{d1}, TS)$ and $S_2(B_1, \ldots, B_{d2}, TS)$, where $\mathcal{A} = \{A_1, \ldots, A_{d1}\} \cup \{B_1, \ldots, B_{d2}\}$ is the set of data attributes of the two streams and TS is the timestamp attribute, (b) sequences of windows, $\ldots W_{1,j} \ldots$ and $\ldots W_{2,j} \ldots$, on the two streams defined by[3] ω and σ, and (c) a join condition, Θ, on the join attribute set $\mathcal{A}_\Theta$, a continuous join-query over data streams seeks to maintain the set of join results $\mathcal{R}_j = W_{1,j} \bowtie_\Theta W_{2,j}$.

4.2.3 Skyline-Window-Joins (SWJ) over Sliding Windows

We refer to the skyline-join operations over sliding windows as *skyline-window-join* (SWJ). A *skyline-window-join* operation over the data streams S_1 and S_2 seeks to maintain, for each window W_j, the subset of tuples in $\mathcal{R}_j$ consisting of tuples not *dominated* by any other tuple(s) in $\mathcal{R}_j$.

Given two tuples p and q, we say that p *dominates* q *in the skyline attribute set*

[3]Note that, while in the more general case the two streams can have different ω and σ, for clarity, we limit the discussion for the case where the two streams have the same window characteristics.

$\hat{\mathcal{A}} \subseteq \mathcal{A}$ *(denoted as*[4] $p \rhd_{\hat{\mathcal{A}}} q$*)*, if

$$\forall X \in \hat{\mathcal{A}}, (p.X \succeq q.X) \wedge (\exists Y \in \hat{\mathcal{A}} \mid p.Y \succ q.Y).$$

In other words, p dominates q if p is better than or equal to ($\succeq$) q in all dimensions and better than ($\succ$) q in at least one dimension of the skyline attribute set $\hat{\mathcal{A}}$.

Definition 5 (Skyline-Window-Join, $\overset{sw}{\bowtie}$)**.** *Given (a) two streams S_1 and S_2, (b) sequences of windows, $\ldots W_{1,j} \ldots$ and $\ldots W_{2,j} \ldots$, on the two streams defined by ω and σ, (c) a join condition, Θ, and (d) a set of skyline attributes, $\hat{\mathcal{A}}$, a tuple, p, is in the j^{th} skyline-join window, $S_1 \overset{sw}{\bowtie}_{\Theta,j,\hat{\mathcal{A}}} S_2$, iff (a) $p \in \mathcal{R}_j = W_{1,j} \bowtie_\Theta W_{2,j}$ and (b) $\nexists q \in \mathcal{R}_j - \{p\}$ s.t. $q \rhd_{\hat{\mathcal{A}}} p$.*

The tuple, p, consists of the concatenation of two tuples p_1 and p_2, where p_1 corresponds to a tuple in stream S_1 and p_2 corresponds to a tuple in S_2. Each skyline point, p, is alive *as long as p_1 and p_2 are* alive *in their respective streams.* ◇

The following is an example of a skyline-window-join query:

Example 8 (SWJ Query)**.** *Given two stock market transaction streams,* `Investment(stockID, price, volume, timestamp)` *and* `Risk(stockID, risk, cost, timestamp)`*, that contain information about stock transactions, the query:*

```
Skyline = SWJ * FROM Investment I, Risk R,
                WHERE I.stockID = R.stockID,
                I.timestamp within last 24h,
                R.timestamp within last 24h,
```

[4]In the rest of the chapter, whenever it is clear from the context, we omit references to $\hat{\mathcal{A}}$ and use $p \rhd q$ to denote that p *dominates* q in the skyline attribute set $\hat{\mathcal{A}}$; also $p \not\rhd q$ indicates that p *does not dominate* q.

```
1/price MAX, volume MAX,
1/risk MAX, 1/cost MAX,
```

equi-joins the pair of streams on the join attributes $\mathcal{A}_\Theta = \{\texttt{Investment.stockID}, \texttt{Risk.stockID}\}$, *within the window constraints of* $\omega = 24$ *hours, and returns results that are not dominated by any other results based on the skyline attributes* $\hat{\mathcal{A}} = \{\texttt{price}, \texttt{volume}, \texttt{risk}, \texttt{cost}\}$. ◇

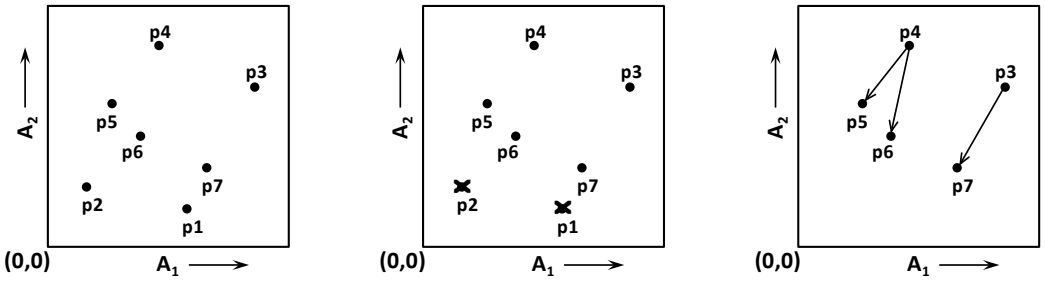
In the above query, the underlying skyline preference function is MAX. Other possible annotations include MIN, where the dimensions are minimized, and DIFF, which denotes that two records with different values in a particular dimension may both be part of the skyline [13]. While any of these monotonic functions are acceptable as preference functions, in the rest of the chapter, without loss of generality, we assume that MAX is specified as the preference function.

4.3 A First Attempt to Processing SWJ Queries

In this section, we first explain *StabSky* [44] and *Iterative* [68] that together form a core part of the novel *iteration-fabric* framework. We then discuss a naive implementation of SWJ, highlight the underlying challenges, and point out the key observations leveraged to develop the `LSJ` algorithm.

4.3.1 The StabSky Algorithm

As mentioned in Section 2.1, *StabSky* [44] is an algorithm that addresses skyline processing over a single data stream under the sliding window model. This algorithm is based on (a) minimizing the number of tuples kept in memory, and (b) effectively characterizing and encoding "critical" dominance relationships among the tuples in the data stream in order to precisely answer all n-of-m skyline queries. Here, m is



(a) Data stream

(b) Redundant tuples $(p1, p2)$; Non-redundant tuples $(p3, \ldots, p7)$

(c) Dominance graph

Figure 4.2: (a) Tuples in the stream arrive the order $p1, p2, \ldots, p7$; (b, c) Data structures used by *StabSky* to process skyline queries over the given stream

the number of most recent tuples seen in a data stream and n ($n \leq m$) is any most recent tuples seen so far.

[44] proves that it is not necessary to maintain all possible dominance relationships among tuples in a data stream to precisely answer n-of-m skyline queries. Thus, it proposes a *dominance graph* – a *forest* structure whose edge set consists of only the *critical* dominance relationships among the tuples in the non-redundant set $\mathcal{N}$. A tuple p is said to be *redundant* with respect to the most recent m elements in the stream if p has expired (i.e. p is outside the most recent m elements) or is dominated by a younger tuple q (i.e. q arrives later than p).

Figure 4.2 illustrates the key features of *StabSky*. Figure 4.2(a) shows a stream of tuples that arrive in the order $p1, p2, \ldots$. In Figure 4.2(b), tuple $p1$ becomes *redundant* since it is dominated by a younger tuple $p7$. The solid black points shown in the figure belong to the set of *non-redundant* tuples, $\mathcal{N}$, and are the only tuples that need to be retained for skyline computation; the remaining tuples ($p1$, $p2$) are considered to be *redundant*.

A dominance relation $q \rightarrow p$ is defined to be *critical* if and only if q is the youngest tuple (but older than p) in $\mathcal{N}$ that dominates p. Figure 4.2(c) visualizes the *dominance*

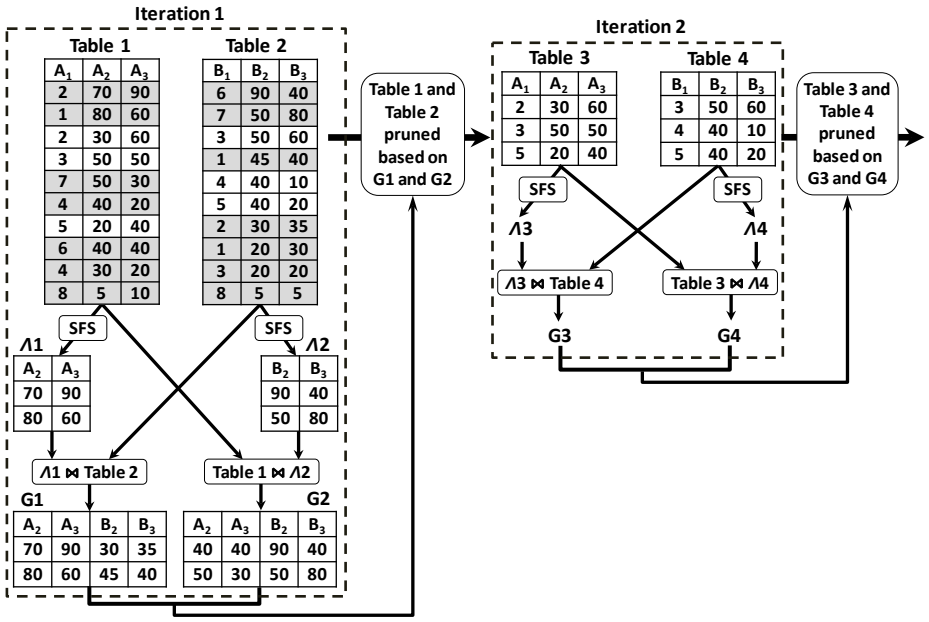


Figure 4.3: Iterative process underlying the existing *skyline-join* operator

graph, $\mathcal{T}$, built on the $\mathcal{N}$ shown in Figure 4.2(b). Here, edge $p4 \rightarrow p6$ is *critical* and the only dominance relation maintained on $p6$ even though it is dominated by both $p3$ and $p4$.

The *StabSky* algorithm uses the above-mentioned data structures to effectively process n-of-m skyline queries. [44] states that for a given n ($n \leq m$), a tuple p in a data stream is a skyline point for the n-of-m query if and only if (a) p is a root node in the current dominance graph, $\mathcal{T}$, or (b) there is a critical dominance edge $q \rightarrow p$ in $\mathcal{T}$ such that q arrives earlier than the n^{th} most recent tuple in the stream.

4.3.2 The Iterative Skyline-Join Algorithm

As discussed earlier, there is a large body of research in the area of static, multi-table skyline-join processing [68, 78, 52]. Among these, Sun *et al.* introduce a new operator called `skyline-join` [68] and two algorithms to support skyline-join queries. The first one extends the *Sort and Limit Skyline (SaLSa)* algorithm [9] to cope with multiple relations. The second algorithm called *Iterative* finds skyline results by

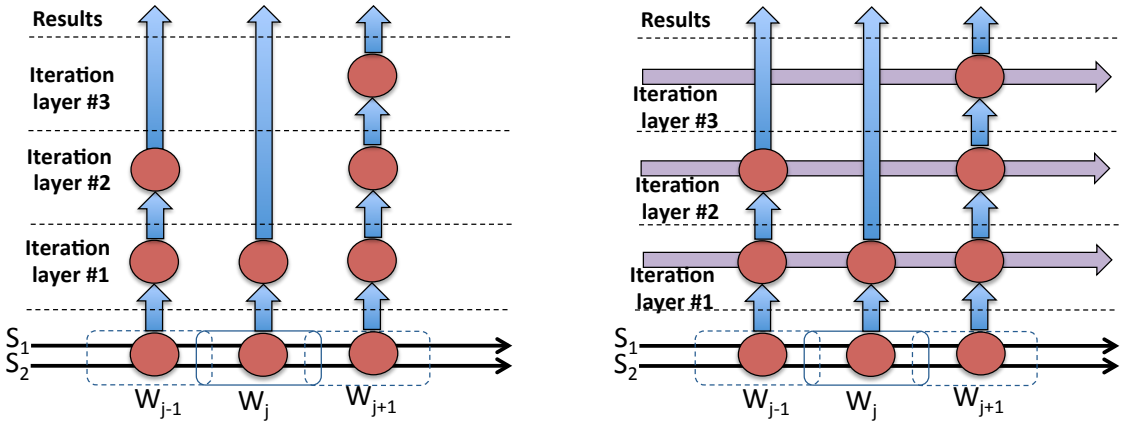


Figure 4.4: (a) Executing skyline-window-join queries by applying the iterative skyline-join algorithm for each window; (b) Viewing layers as separate "virtual streams" that feed the upper layers of iteration

pruning the search space iteratively. Overall, the *Iterative* algorithm is shown to perform well.

Figure 4.3 illustrates the *Iterative* algorithm. Here, Table 1 and Table 2 are the input tables, A_2, A_3, B_2, B_3 represent the skyline attributes, and the join is carried out on A_1 and B_1. *Iterative* first computes the local skyline ($\Lambda 1$, $\Lambda 2$) of the input tables. It then generates skyline results ($\mathcal{G}1$, $\mathcal{G}2$) of the skyline-join using $\Lambda 1$ and $\Lambda 2$. These results are used for pruning the input tables to obtain Table 3 and Table 4; pruned tuples are highlighted in Figure 4.3. This completes one iteration. The process is then repeated with Table 3 and Table 4 being used as inputs to the next iteration. This process continues until at least one of the input tables is completely eliminated.

4.3.3 Key Insights: Layers and Overlaps

It is easy to see that a simple way to execute skyline-window-join operations is to apply the iterative skyline-join algorithm for each window. Figure 4.4(a) visualizes the process. More importantly, however, we observe, that the consecutive iterations

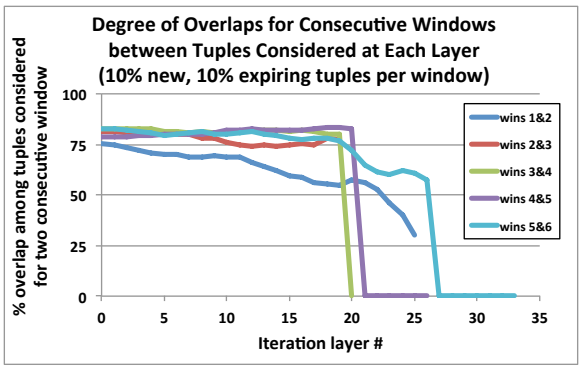


Figure 4.5: Sample SWJ execution for 5 consecutive windows (10% new and 10% expiring tuples per window): the plots show that the skyline-join process iterate somewhere between 20 to 35 times for different windows and the overlaps (among consecutive windows) of tuples considered at different layers of iterations remain high across layers of iteration

of the algorithm, spanning multiple windows, can be viewed as separate *iteration layers* (Figure 4.4(b)).

> Our key insight in this work is that *overlaps exist not only at the lowest data layer (across consecutive data windows), but also at the individual iteration layers, where the tuples processed can be considered as "virtual streams" that evolve from one window to the next (see Figure 4.5 for a sample execution).*

Therefore, we argue that if we naively execute the skyline-window-join operation by applying the iterative skyline join algorithm separately for each window, we can end up with significant amount of redundant work. We further argue that if we can quickly identify and eliminate these per-layer overlaps, we can achieve significant savings in processing time.

Based on these insights, in the next section, we present a novel Layered Skyline-window-Join (`LSJ`) operator which avoids redundancies in skyline-join query processing by weaving together the consecutive windows and consecutive iteration layers into an *iteration-fabric* processing structure.

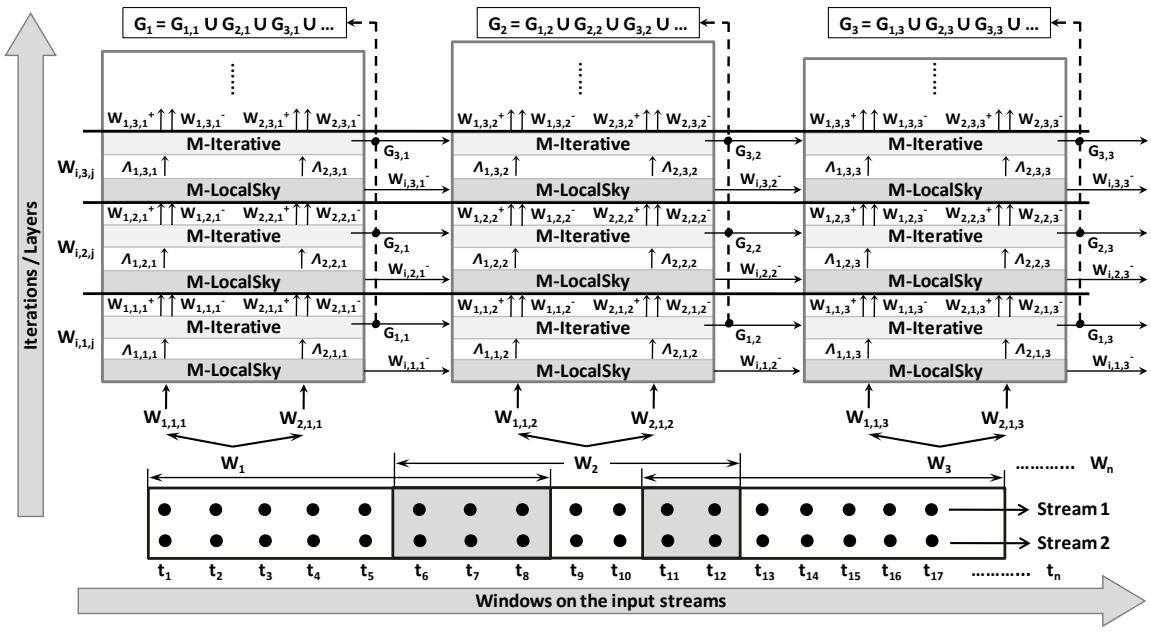


Figure 4.6: Overview of the *iteration-fabric*, which weaves together layer-modules (or $\mathcal{L}$-modules, each consisting of two sub-modules, `M-LocalSky` and `M-Iterative`) into a grid across iterations and windows

4.4 Layered Skyline-Window-Join (LSJ) Operator

In this section, we present an efficient Layered Skyline-window-Join (`LSJ`) operator for computing skyline-joins over sliding windows. `LSJ` leverages a novel *iteration-fabric* processing structure to identify and eliminate per-layer overlaps across consecutive windows.

4.4.1 Iteration-Fabric Processing Structure

Figure 4.6 gives an overview of the proposed *iteration-fabric* framework. As shown in the figure, for each window, `LSJ` applies an iterative skyline-join process: the data is passed to the lowest layer-module (or $\mathcal{L}$-module), where each $\mathcal{L}$-module

1. computes the local skylines of the input windows,

2. generates partial skyline-join results,

3. prunes the tuples in the input windows using the partial skyline-join result set, and lastly,

4. passes the pruned data to the $\mathcal{L}$-module at the next layer.

It is important, however, to note that the $\mathcal{L}$-modules are not only vertically connected across different layers of the same window, but also horizontally connected across consecutive windows of the same layer. This enables the iteration-fabric structure to identify and eliminate processing redundancies across all layers of the skyline-window-join processing[5].

As shown in Figure 4.6, each $\mathcal{L}$-module consists of two sub-modules, `M-LocalSky` and `M-Iterative`.

The `M-LocalSky` Sub-Module

At layer, l, of the j^{th} window, the `M-LocalSky` sub-module produces the local skyline sets, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$, of the tuples corresponding to the data sets $W_{1,l,j}$ and $W_{2,l,j}$, corresponding to the layer l of the j^{th} window, obtained from the $\mathcal{L}$-module at the $(l-1)^{th}$ layer of the j^{th} window.

As described in Section 4.3.2, each iteration of the *Iterative* algorithm [68] starts with finding the local skyline points of the input data to that iteration. The authors use the well-known *Sort-Filter-Skyline (SFS)* algorithm [15] to compute the local skyline points at each iteration. SFS, however, is applicable to scenarios in which the data is static. Therefore, while we can use SFS to compute local skylines only based on inputs in a given window, this would not enable us to leverage the overlaps in the data at the same layer in consecutive windows.

[5]This both horizontally and vertically connected iteration-fabric structure also provides opportunities for highly-parallel executions where different $\mathcal{L}$-modules are associated to different processing units (e.g., cores). We leave the investigation of this to future work.

Instead, `M-LocalSky` generates the local skyline points, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$, not only by considering $W_{1,l,j}$ and $W_{2,l,j}$ passed from a lower layer, but also $W_{1,l,j-1}$ and $W_{2,l,j-1}$ from the previous window. Intuitively, tuples in $\Delta^{-}_{i,l,j} = W_{i,l,j-1} \backslash W_{i,l,j}$ are considered as the expiring tuples at the window j of layer l and $\Delta^{+}_{i,l,j} = W_{i,l,j} \backslash W_{i,l,j-1}$ are treated as the new tuples at the window j of layer l; here "\" is the set difference operator. Given these, we compute, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$, not using using a static skyline technique like SFS, but relying on a modified version of the on-line skyline techniques (*StabSky*) presented in [44]. Specifically, we adapt the methods proposed for processing n-of-m skyline queries over count-based sliding windows (see Section 4.3.1). In essence, the data $W_{1,l,*}$ and $W_{2,l,*}$ at layer l are treated as streams that evolve over time, and their local skyline sets at the j^{th} window (i.e, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$) are computed by taking into account the overlaps between the same layer of the *previous* and the current window being analyzed. The details of `M-LocalSky` are presented in Section 4.4.2.

The `M-Iterative` Sub-Module

The local skyline sets produced by `M-LocalSky` are pushed into the `M-Iterative` sub-module that uses an adapted version of *Iterative* [68] to produce the global skyline set, $\mathcal{G}_{l,j}$, of the layer l in the current window j. Once $\mathcal{G}_{l,j}$ is identified, `M-Iterative` then prunes the input data sets $W_{1,l,j}$ and $W_{2,l,j}$ and pushes the tuples that qualify to the next layer, $l+1$, as $W_{1,l+1,j}$ and $W_{2,l+1,j}$. The overall global skyline set, $\mathcal{G}_j$, of the skyline-join query over the current sliding window j is given by the union of the global skyline sets produced at each layer of the window. We discuss the details of the `M-Iterative` sub-module in Section 4.4.3.

4.4.2 Local Skyline Computation Module (M-LocalSky)

We build the M-LocalSky component of the *iteration-fabric* based on a modified version of the *StabSky* algorithm [44], which maintains skylines of data streams that evolve over time. The proposed M-LocalSky module (Figure 4.7) computes the local skyline sets ($\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$) by taking into account the overlaps between the same layer of the *previous* and the current window being analyzed.

M-LocalSky implements a time-based sliding window version of the *StabSky* algorithm. This is achieved by replacing a tuple p's position label by its generation timestamp $p.start$. M-LocalSky maintains *non-redundant* tuples ($\mathcal{N}_{i,l,j}$) for each layer l of stream $i \in \{1, 2\}$. It also maintains the corresponding *dominance graph* ($\mathcal{T}_{i,l,j}$) for each layer l built on the non-redundant tuples in $\mathcal{N}_{i,l,j}$. The dominance graphs are encoded into *intervals* and stored in *interval trees* using the scheme described in [44]. For example, the dominance graph in Figure 4.2(c) built on the tuples $p1, p2, p3, \ldots$ with generation timestamps $1, 2, 3, \ldots$, respectively, can be encoded as the intervals $(0, 3]$, $(0, 4]$, $(3, 7]$, $(4, 5]$, and $(4, 6]$.

As shown in Figure 4.7, given the set of expiring ($\Delta^{-}_{i,l,j}$) and new tuples ($\Delta^{+}_{i,l,j}$) of window j at layer l, M-LocalSky makes modifications to the *non-redundant* tuple set ($\mathcal{N}_{i,l,j}$) and the *dominance graph* ($\mathcal{T}_{i,l,j}$) of each layer l in stream $i \in \{1, 2\}$ by applying a slightly different version of the techniques developed in [44]. In the original *StabSky* algorithm, when a new tuple, p_{new}, arrives in the data stream, p_{new} is added to the set of *non-redundant* tuples, $\mathcal{N}$, and the oldest element, p_{old}, in $\mathcal{N}$ is removed only if it has expired. The M-LocalSky module, on the other hand, makes additions and deletions of tuples to the *non-redundant* sets and *dominance graphs* based on $\Delta^{-}_{i,l,j}$ and $\Delta^{+}_{i,l,j}$.

At the end of each call, M-LocalSky returns the local skyline points ($\Lambda_{i,l,j}$) for

Algorithm 1: M-LocalSky($\Delta^-_{i,l,j}$, $\Delta^+_{i,l,j}$, $\hat{\mathcal{A}}$)

Input:

$\Delta^-_{i,l,j}$: Expiring tuples of window j at layer l for stream $i \in \{1, 2\}$

$\Delta^+_{i,l,j}$: New tuples of window j at layer l for stream $i \in \{1, 2\}$

$\hat{\mathcal{A}}$: Skyline attribute set of stream $i \in \{1, 2\}$

Output:

$\Lambda_{i,l,j}$: Local skyline set of layer l for stream $i \in \{1, 2\}$ in window j

Procedure:

for each expired tuple p_{old} in $\Delta^-_{i,l,j}$ **do**

$\mathcal{N}_{i,l,j} := \mathcal{N}_{i,l,j} - \{p_{old}\}$

remove interval $(a.start, p_{old}.start]$ with $p_{old}.start$ as the right end from $\mathcal{T}_{i,l,j}$

for each old critical edge $p_{old} \rightarrow p$ **do**

find the new critical edge $a \rightarrow p$

update interval $(p_{old}.start, p.start]$ in $\mathcal{T}_{i,l,j}$ to $(a.start, p.start]$

(or $(0, p.start)$])

end for

end for

for each new tuple p_{new} in $\Delta^+_{i,l,j}$ **do**

find $D_{p_{new}} \subseteq \mathcal{N}_{i,l,j}$ dominated by p_{new}

for each tuple $p \in D_{p_{new}}$ **do**

remove the intervals in $\mathcal{T}_{i,l,j}$ with $p.start$ as an end

end for

$\mathcal{N}_{i,l,j} := \mathcal{N}_{i,l,j} - D_{p_{new}} + \{p_{new}\}$

find the critical edge $p \rightarrow p_{new}$

add $(p.start, p_{new}.start]$ (or $(0, p_{new}.start)$]) to $\mathcal{T}_{i,l,j}$

end for

add the root nodes of $\mathcal{T}_{i,l,j}$ to $\Lambda_{i,l,j}$

return $\Lambda_{i,l,j}$

Figure 4.7: The `M-LocalSky` module

Algorithm 2: M-Iterative($W_{i,l,j}$, $\Lambda_{i,l,j}$, $\hat{\mathcal{A}}$)

Input:

$W_{i,l,j}$: Data for stream $i \in \{1,2\}$ at layer l in window j

$\Lambda_{i,l,j}$: Local skyline set of layer l for stream $i \in \{1,2\}$ in window j

$\hat{\mathcal{A}}$: Skyline attribute set of stream $i \in \{1,2\}$

Output:

$\mathcal{G}_{l,j}$: Global skyline set of layer l in window j

Procedure:

$P_1 = \Lambda_{1,l,j} \overset{sw}{\bowtie} W_{2,l,j}$

$P_2 = W_{1,l,j} \overset{sw}{\bowtie} \Lambda_{2,l,j}$

$\mathcal{G}_{l,j} = \mathcal{G}_{l,j} + P_1 + P_2$

$\mathcal{G}_j = \mathcal{G}_j + \mathcal{G}_{l,j}$

Prune $W_{i,l,j}$ using *outsiderPrune* [68]

Push unpruned tuples in $W_{i,l,j}$ to iteration layer $W_{i,l+1,j}$

return $\mathcal{G}_{l,j}$

Figure 4.8: The `M-Iterative` module

each stream $i \in \{1,2\}$. The local skyline sets contain only the root nodes of the corresponding dominance graphs since we consider the sliding window model. The sliding window model is a special case of the n-of-m query model described in [44]; in sliding windows $n = m$.

4.4.3 Iteration Module (`M-Iterative`)

The proposed `M-Iterative` module is described in Figure 4.8. At each call of `M-Iterative`, the local skyline sets ($\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$) generated by the `M-LocalSky` module are processed to obtain the global skyline set ($\mathcal{G}_{l,j}$) of layer l in window j. In the modified version of the original *Iterative* algorithm [68], the `M-Iterative` module performs window-based joins using the symmetric hash join method [81].

Once $\mathcal{G}_{l,j}$ is computed, `M-Iterative` prunes the input windows, $W_{1,l,j}$ and $W_{2,l,j}$, using *outsiderPrune*[6] [68]. Finally, `M-Iterative` completes execution by pushing the unpruned tuples to the next layer, $l+1$, as $W_{1,l+1,j}$ and $W_{2,l+1,j}$. The overall global skyline, $\mathcal{G}_j$, of the SWJ query over the current sliding window j is incrementally maintained by `M-Iterative`; $\mathcal{G}_j$ is obtained by the union of the global skyline sets produced at each layer l of window j.

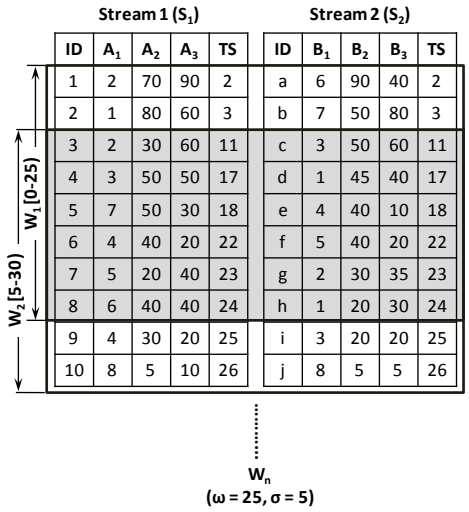
4.4.4 Truncated Layered Skyline-Window-Join

Let us consider Figure 4.5 which shows a sample execution of a SWJ query for 5 consecutive windows, with 10% new tuples arriving and 10% tuples expiring per window. It is easy to see from the figure that the per-layer overlaps tend to drop as the iteration count increases: there are often larger number of overlaps in the earlier iterations, whereas the number of overlaps gets almost monotonically lower as the iterations progress. Since the benefits of the *iteration-fabric* depend on the degree of per-layer overlap, it may be advantageous to stop checking for overlaps against the previous window once the degree of overlap drops below a preset threshold at an iteration level or when the gains achieved through overlap analysis falls below the time needed to maintain the data structures. We refer to this as *truncated Layered Skyline-window-Join (*`LSJ`*)* processing.

4.4.5 Example Execution of Layered Skyline-Window-Joins

Figure 4.9 illustrates the proposed `LSJ` operator. The incoming streams are bound by sliding windows (Figure 4.9(a)) and skyline-window-joins are executed over these

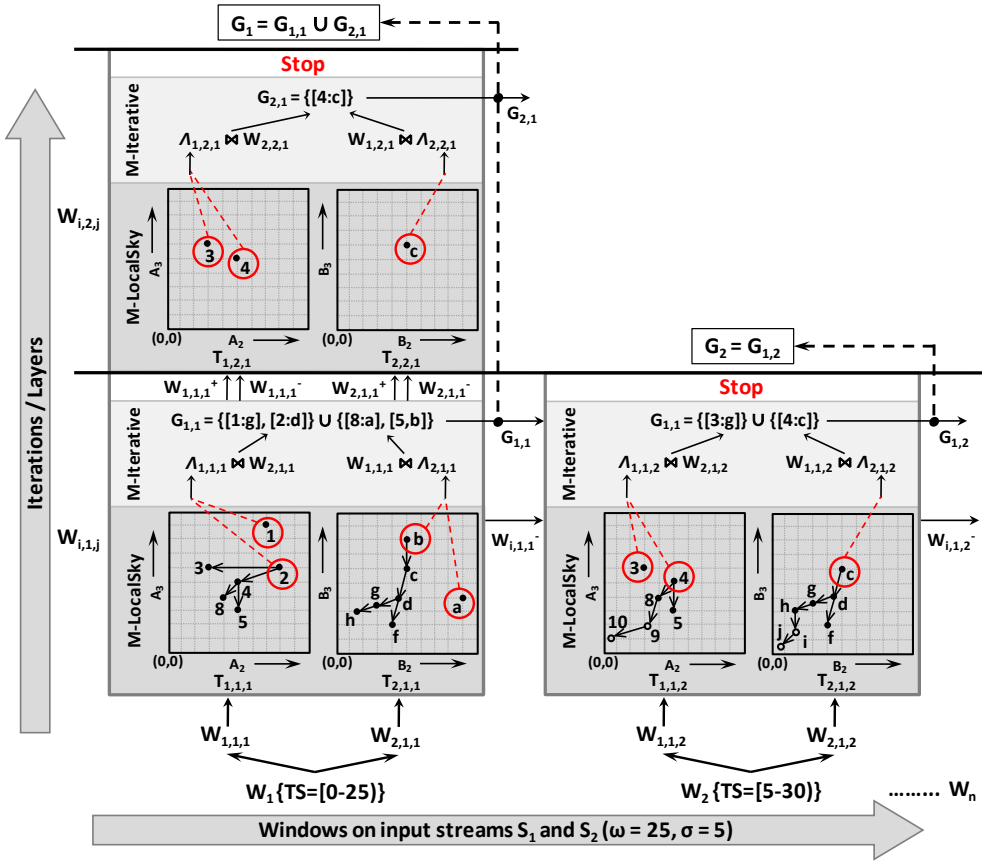
[6]The *APDominatePrune* optimization in [68] adds significant run-time execution cost and hence is not suitable for streaming scenarios; thus the methods described in this work do not use this optimization.



ID	A_1	A_2	A_3	TS
1	2	70	90	2
2	1	80	60	3
3	2	30	60	11
4	3	50	50	17
5	7	50	30	18
6	4	40	20	22
7	5	20	40	23
8	6	40	40	24
9	4	30	20	25
10	8	5	10	26

ID	B_1	B_2	B_3	TS
a	6	90	40	2
b	7	50	80	3
c	3	50	60	11
d	1	45	40	17
e	4	40	10	18
f	5	40	20	22
g	2	30	35	23
h	1	20	30	24
i	3	20	20	25
j	8	5	5	26

(a) Input streams bound by windows



(b) Execution of skyline-window-joins using the *iteration-fabric* framework

Figure 4.9: An example Layered Skyline-window-Join (LSJ) operation

sliding windows using the novel *iteration-fabric* framework (Figure 4.9(b)).

In the given example, S_1 (containing tuples with IDs $1, 2, \ldots$) and S_2 (containing tuples with IDs $a, b, \ldots$) are the input streams, $\hat{\mathcal{A}} = \{A_2, A_3, B_2, B_3\}$ is the set of skyline attributes, TS is the timestamp attribute, and an equi-join is carried out on the join attribute set $\mathcal{A}_\Theta = \{A_1, B_1\}$. The input streams are bound by a sliding window of size $\omega = 25$. Each window W_j spans a time period of $[W_j.start, W_j.end)$, where $W_j.end = W_j.start + \omega$; for instance, the time period of $W_1 = [0, 25)$. The sliding window is constrained by $\sigma = 5$, i.e. the start of the window shifts 5 units at every window update; for example, $W_2.start = W_1.start + \sigma = 5$.

`LSJ` executes SWJ queries by pushing the tuples contained in a sliding window through the `M-LocalSky` and `M-Iterative` modules. As shown in Figure 4.9(b), `LSJ` starts with window W_1 and sends the tuples of each of the streams in layer 1 ($W_{1,1,1}$, $W_{2,1,1}$) to the `M-LocalSky` module in order to obtain the local skyline sets $\Lambda_{1,1,1}$ and $\Lambda_{2,1,1}$. Since this is the very first window being executed, `M-LocalSky` does not have to look at the layers of a previous window, and therefore, no overlap analysis is performed at this stage. `M-LocalSky` builds the dominance graphs $\mathcal{T}_{1,1,1}$ and $\mathcal{T}_{2,1,1}$ on the non-redundant tuple sets $\mathcal{N}_{1,1,1} = (1, 2, 3, 4, 5, 8)$ and $\mathcal{N}_{2,1,1} = (a, b, c, d, f, g, h)$, respectively. The roots of $\mathcal{T}_{1,1,1}$ and $\mathcal{T}_{2,1,1}$ are the local skyline points of layer 1 in W_1, i.e. $\Lambda_{1,1,1} = (1, 2)$ and $\Lambda_{2,1,1} = (a, b)$. These local skyline sets are then pushed to the `M-Iterative` module.

`M-Iterative` generates the global skyline of layer 1 in W_1 ($\mathcal{G}_{1,1}$) using the local skyline sets obtained from `M-LocalSky`. $\mathcal{G}_{1,1}$ is then used for pruning $W_{1,1,1}$ and $W_{2,1,1}$ to obtain the tuples that qualify to be in layer 2 of W_1 ($W_{1,2,1}$, $W_{2,2,1}$). As seen in Figure 4.9(b), the dominance graphs $\mathcal{T}_{1,2,1}$ and $\mathcal{T}_{2,2,1}$ of layer 2 in W_1 reflect the tuples pruned. `LSJ` repeats its calls to `M-LocalSky` and `M-Iterative` as before; it eventually comes to a stop when all tuples in the current window are pruned and no new layers

exist. The global skyline set of W_1 is given by the global skyline sets obtained at every layer, i.e. $\mathcal{G}_1 = \mathcal{G}_{1,1} + \mathcal{G}_{2,1}$.

After analysing W_1, `LSJ` continues execution on window W_2 (obtained based on the parameters $\omega = 25$ and $\sigma = 5$). Once again, `LSJ` sends the tuples in W_2 to the `M-LocalSky` and `M-Iterative` modules to obtain the global skyline of this window. The key difference here is that the `M-LocalSky` module now analysis the overlap between the layers of the current and previous windows. Figure 4.9(a) shows that there is a significant overlap between windows W_1 and W_2. Therefore, when the tuples of each of the streams in a particular layer of W_2 are pushed to the `M-LocalSky` module, it makes insertions and deletions to the dominance graphs only based on the set of expiring and new tuples – tuples that overlap are not reprocessed. For instance, in Figure 4.9(b), the overlap set between layers $W_{1,1,1}$ and $W_{1,1,2}$ contains tuples with IDs $3, 4, 5, 8$ (indicated by the solid dots in dominance graph $\mathcal{T}_{1,1,2}$), so the set of new tuples contains $9, 10$ and the set of expiring tuples has $1, 2$. Thus, based on these sets of expiring and new tuples, the dominance graph of Stream 1 in layer 1 of W_2 ($\mathcal{T}_{1,1,2}$) is obtained by making insertions and deletions to the previous dominance graph of Stream 1 in layer 1 of W_1 ($\mathcal{T}_{1,1,1}$). If a previous layer doesn't exist, then the `M-LocalSky` module processes the current layer similar to how it processes layers in window W_1 (described earlier).

Note that, in this simplified example, sharing across windows is leveraged only at the first layer. However, in practice there exists opportunities for sharing at more than one layer (see Figure 4.5), and as evaluated in the next section, `LSJ` leverages these overlaps for improved performance.

4.5 Performance Evaluation

In this section, we evaluate the performance of `LSJ` on real and simulated data sets by varying the parameters involved.

4.5.1 Experimental Setup

This is the first research work that we are aware of which targets skyline-join queries on data streams. Therefore, for comparison purposes, we have implemented alternative schemes, with varying degrees of *intelligence*:

- A `Naive` method which first performs a window-based join of the input streams using the symmetric hash join method [81]. The join results are then processed using SFS [15] to obtain the final skyline set of each window.
- An *iterative-skyline-join* (`ISJ`) scheme that operates by applying SFS at each iteration (Section 4.3.2).
- In addition, we have varied the *truncation layer*, l, of the *iteration-fabric*. Here, `LSJ` ($l = *$) means that the first $*$ layers of a particular sliding window are processed using the *iteration-fabric* processing structure (Section 4.4.1), while the remaining layers in the window are executed using the *iterative-skyline-join* (`ISJ`) scheme (Section 4.3.2). `LSJ` ($l = 1$) corresponds to the case where the overlaps only at the data-entry level are leveraged using *StabSky* [44], whereas `LSJ` and `LSJ` (*all*) correspond to the case where overlaps are identified and leveraged at all levels of the *iteration-fabric*.

• **Evaluation Platform.** The above algorithms were all implemented in Java. The Interval tree used for maintaining dominance graphs in `M-LocalSky` was adapted from

Table 4.2: Intel Berkeley research lab dataset

date:yyyy-mm-dd	time:hh:mm:ss.xxx	epoch:int	moteid:int
temperature:real	humidity:real	light:real	voltage:real

`www.gephi.org`[7]. All experiments were conducted on a machine with an Intel Core i5 3.10GHz processor, 8GB RAM (1GB of which is available to the Java machine[8]) and Windows 7 operating system. Each experiment is run three times and the execution times reported are the averages of the three runs.

• **Datasets.** The evaluations were carried out on both synthetic and real data. Synthetic streams were generated based on correlated, independent and anti-correlated distributions[9] as described in [13]. Since the tuples generated by [13] have no timestamps, we borrowed the *epoch* values from the *Intel Berkeley Research lab* data set[10] and used them as timestamps.

We also ran experiments on the above-mentioned *Intel Berkeley Research (IBR) lab* sensor data stream. This data contains readings collected from 54 sensors and has about 2.3 million readings. The complete schema of the data set is shown in Table 4.2. In this, *moteid* is a unique integer assigned to each sensor and *epoch* corresponds to a monotonically increasing sequence number obtained from the sensors every 30 seconds. The data set also gives the x and y coordinates of the sensors We use this information to calculate each sensor's distance from the point $(0, 0)$ and utilize this as an additional attribute.

• **Evaluation Parameters.** As is common in assessing skyline algorithms, we use execution time as the major evaluation metric. Execution time of a SWJ query is the

[7]Source code available at `https://github.com/gephi/gephi-launchpad-branches/tree/master/AttributesAPI/src/org/gephi/data/attributes/type`.

[8]In these experiments, the memory was not a bottleneck.

[9]`http://randdataset.projects.postgresql.org/`.

[10]`http://db.csail.mit.edu/labdata/labdata.html`.

duration from the time an algorithm starts to the time it returns the entire skyline set. It also includes the time taken to maintain the various index structures that are built on-the-fly. The streams were analyzed over 30 sliding windows and the execution time gains relative to competitor methods are reported. Time gain (%) is calculated as $(1 - a/b) \times 100$, where a, b represent the total time taken to execute skyline-joins over 30 windows by the two algorithms being compared (i.e. a vs. b).

The analysis was carried out over windows of different sizes (ω) and window shift lengths (σ). We also analyzed the effects of the join rate (r) between the streams and the dimensionality (d) of the skyline attribute set per input stream.

4.5.2 Evaluation over Real Streams

This section presents evaluation results of the proposed approaches on real data. We consider a scenario where the even-numbered *IBR lab* sensors produce readings only related to *temperature* and *voltage*, while the odd-numbered sensors give readings of *humidity* and *light*. This results in two input streams, *intel-even (moteid, temperature, voltage, epoch, distance)* and *intel-odd (moteid, humidity, light, epoch, distance)*. Given these, we search for the skyline-join over the set of sensors that are distance, δ, from the origin $(0, 0)$ of the room for the attributes *temperature, voltage, humidity, light*. Informally, this query returns a set of *interesting* readings produced by sensors that belong to a particular region of the room. The sliding window is defined using the *epoch* attribute.

• **Impact of the Number of Layers (l) on `LSJ`.** Figure 4.10 illustrates the behavior of `LSJ` as the number of layers (l) to which the *iteration-fabric* is applied changes. Here, the SWJ operation is carried out among sensors that are at a distance of 20 meters from the corner of the room.

The figure shows that the execution time gain achieved by `LSJ` increases as l in-

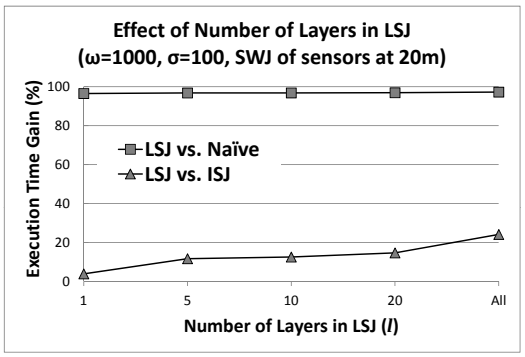
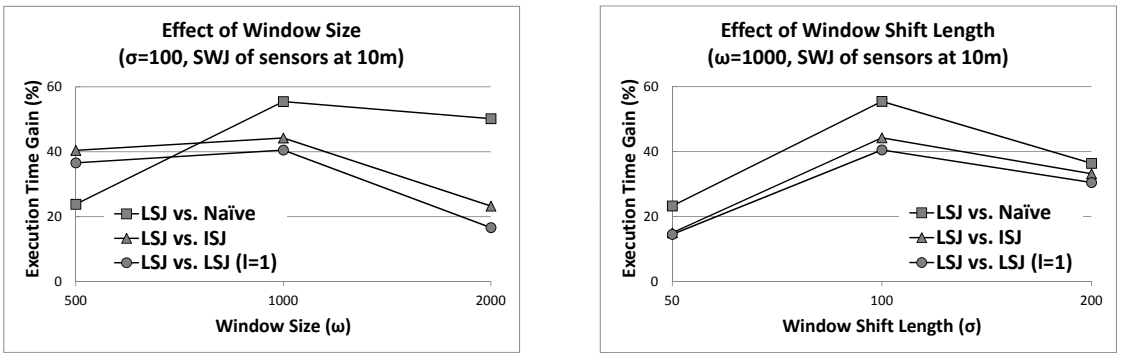


Figure 4.10: Effect of number of layers (l) in `LSJ` (Since `ISJ` does not consider layer overlaps, l has no impact on its execution time)



(a) Effect of window size (ω)

(b) Effect of window shift length (σ)

Figure 4.11: Evaluation over real streams

creases, with the case where *iteration-fabric* is applied to all layers (`LSJ` ($l = all$)) performing the best overall. Figure 4.10 shows that, for the given configuration, `LSJ` has significant ($> 95\%$) gains over `Naive`. More importantly, however, the performance of `LSJ` increases with the number layers included in the *iteration-fabric* and it provides $\sim 25\%$ gain over `ISJ`, when all the layers are included.

As we see in the rest of the experiments, this gain is a function of the various parameters, including amount of data and the data distribution and in our experiments with this real data set, the gains over `ISJ` varied between $\sim 25\%$ and $\sim 40\%$. Gains up to $\sim 80\%$ are observed in correlated data streams as discussed in Section 4.5.3.

• **Effect of the Window Size.** The experiment reported in Figure 4.11(a) studies

the effect of the size of the sliding windows (ω) – for a given window shift. In this setup, the SWJ operation is carried out among sensors that are at a distance of 10 meters from the corner of the room. Figure 4.11(a) shows that `LSJ` performs better than `Naive`, `ISJ`, and `LSJ` ($l = 1$) for varying window sizes. The performance of `LSJ` is especially high ($\sim 45\%$ gain over `ISJ`) when the window size is $10\times$ the shift length. The time gains relative to `ISJ` and `LSJ` ($l = 1$) drop when the window size is much larger. This means that, for this data, the overlaps at higher iteration layers are not large enough to compensate for the overhead of maintaining the necessary data structures. Thus, these data streams would benefit from a truncated execution of `LSJ` (described in Section 4.4.4).

• **Effect of Window Shift Length.** Figure 4.11(b) examines the execution time gains achieved by `LSJ` as the window shift length (σ) is changed – for a fixed window size. A larger window shift length implies a bigger change in the layers of the *iteration-fabric*, with lesser overlaps being present between the layers of consecutive windows. As shown in Figure 4.11(b), `LSJ` successfully leverages any overlaps that may exist and executes SWJ queries faster, more efficiently than the `Naive`, `ISJ`, and `LSJ` ($l = 1$) methods and provides high ($\sim 45\%$) gains over `ISJ` when the window size is $10\times$ the shift length. Note that there is a drop in performance gain when the shift length is very small ($\sigma = 50$). As we later experimentally show in Section 4.5.3, this occurs when the skyline attributes are relatively independent. Once again, in such a scenario a truncated execution of `LSJ` would be beneficial.

4.5.3 Evaluation over Synthetic Streams

In this section, we carry out a more detailed analysis of `LSJ` on synthetically generated data streams, where we also vary the join selectivity and consider correlated, independent and anti-correlated distributions. The default dimensionality (d) of the

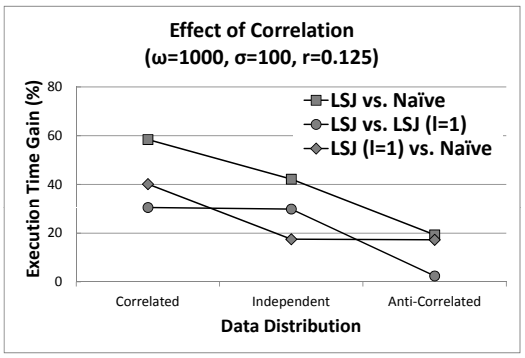


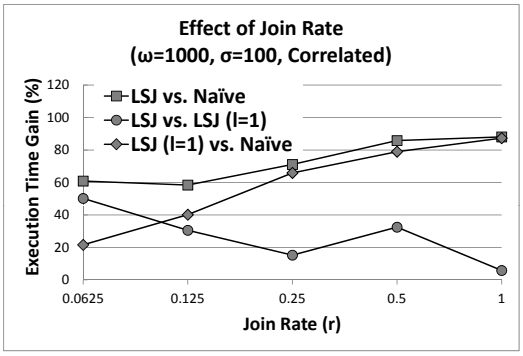
Figure 4.12: Effect of data correlation

skyline attribute set per data stream is set to 2, which gives a total of 4 skyline attributes for SWJ operations.

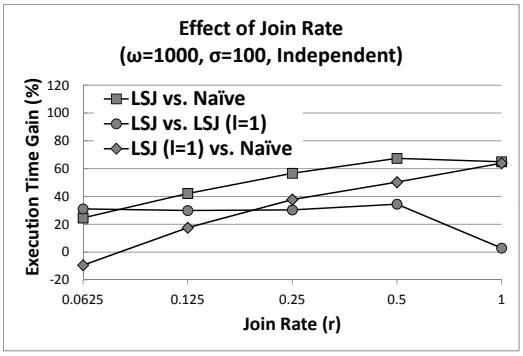
• **Effect of Correlation.** Figure 4.12 illustrates that `LSJ` provides the largest gains on correlated data distributions and provides the lowest gains in anti-correlated distributions. In fact, `LSJ`'s (i.e., `LSJ` (all)'s) performance gains against `LSJ` ($l = 1$) is close to 0%. This implies that in anti-correlated data sets, the process does not generate sufficiently large overlaps in high-numbered iteration layers – most of the overlaps are identified and eliminated at the data layer itself; therefore, `LSJ` ($l = 1$) itself is sufficient. In correlated and independent data sets, however, eliminating overlaps at the higher numbered iteration layers provide $\sim$ 30% execution time gains over `LSJ` ($l = 1$).

Note that both `LSJ` and `LSJ` ($l = 1$) perform better for correlated data than for anti-correlated data streams. This is expected as the *Iterative* technique that forms the basis of `LSJ` family of algorithms is known to perform less efficiently on anti-correlated data distributions [68].

• **Effect of Join Rate.** Figure 4.13 shows the effect of the join rate (r) between pairs of sliding windows. We can observe that as the join work increases with the join rate, `LSJ`'s embedding of the join computation into the skyline process provides larger gains during SWJ operations.



(a) Correlated

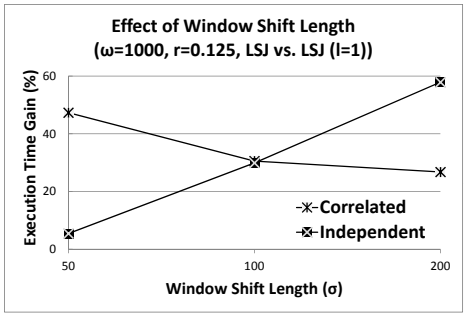


(b) Independent

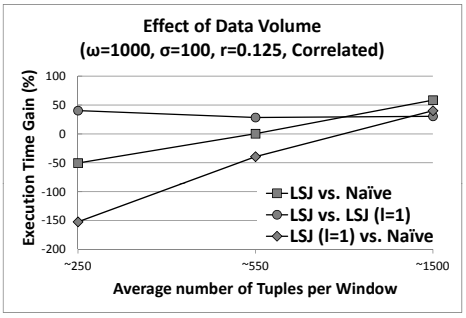
Figure 4.13: Effect of join rate (r)

However, it is interesting that, as can be seen in Figure 4.13(a), on correlated data sets, `LSJ`'s gains over `LSJ` ($l = 1$) drops as the join rate increases – in contrast, on independent data sets (Figure 4.13(b)), the gain relative to `LSJ` ($l = 1$) stays constant. This implies that on correlated data sets most of the overlaps are identified and removed early in the iteration process, whereas in independent data sets, this is not the case.

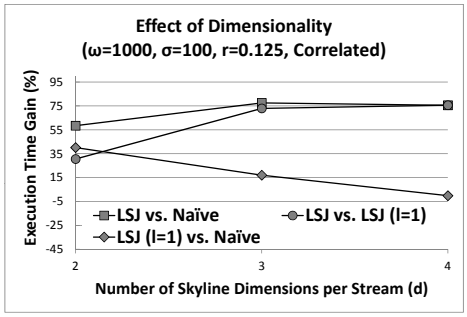
• **Effect of Window Shift Length.** Figure 4.14(a) examines the time gains achieved by the proposed `LSJ` method as the window shift length (σ) is changed. As mentioned before, a larger window shift length implies a bigger change in the layers of the *iteration-fabric*, with lesser overlaps being present between the layers of consecutive windows. As seen in Figure 4.14(a), the performance also depends on data distribution. For correlated data, as expected, higher overlaps (smaller shift lengths) provide larger gains. For independent data, however, the behavior is the opposite. This implies that in skyline-window-joins with independent skyline attributes most of the overlaps are eliminated in the small-numbered (i.e. earlier) iteration layers. This highlights that a truncated execution of `LSJ` would be useful in cases where the data streams are not highly correlated.



(a) Effect of window shift length (σ)



(b) Effect of data volume



(c) Effect of dimensionality (d) of skyline attribute set per input stream

Figure 4.14: Evaluation over synthetic streams

• **Effect of the Data Volume/Window Size.** This experiment (Figure 4.14(b)) examines the effect of the changes in the volume of data contained in each sliding window. As can be observed, when the the data volume is low, the overhead of processing data through the *iteration-fabric* is not worthwhile. However, as the volume of the data increases, the benefits of the `LSJ` approach becomes more and more apparent.

• **Effect of Skyline Dimensionality.** Figure 4.14(c) shows that `LSJ` scales particularly well to SWJ operations on high dimensional skyline queries. As can be observed, the gains of `LSJ` increases both against `Naive` and `LSJ` ($l = 1$) as the number of skyline dimensions increases. Hence, we can conclude that as the number of skyline dimensions increases, less overlaps are removed at earlier iterations and thus applying the *iteration-fabric* to all layers of SWJ brings better performance.

Chapter 5

EFFICIENT PROCESSING OF STRATA-SKYLINE QUERIES OVER INCOMPLETE DATA SOURCES

5.1 Introduction

Many of the early skyline algorithms [13, 37, 15, 55, 39] assumed that all skyline attribute values are precise and available. Unfortunately, data imprecision is ubiquitous in many applications (such as traffic monitoring, sensor networks, environmental surveillance, and location-based analysis) where imprecision is caused by incompleteness, human errors, or imperfections in data capture. Naturally, such imprecisions render basic skyline algorithms unsuitable for use.

5.1.1 Data Incompleteness

In this work, we focus on *incomplete* data, where some of the attribute values of some of the tuples are missing (and are represented by NULL values). This can be because of various reasons, such as data-entry errors, lack of knowledge, and privacy. For instance, if we consider a set of movies rated by different users, many ratings may be missing as most often users do not rate movies that they have not seen. These unknown movie ratings would be described as missing or NULL values. Figure 5.1 illustrates such a scenario. In this figure, the horizontal line, b, represents an incomplete tuple, since it has a missing value for the *Movie Rating* attribute.

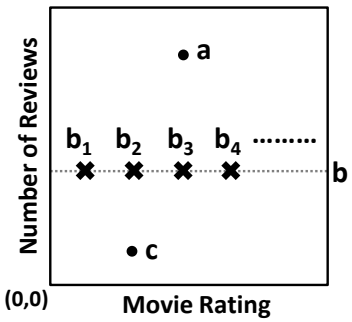
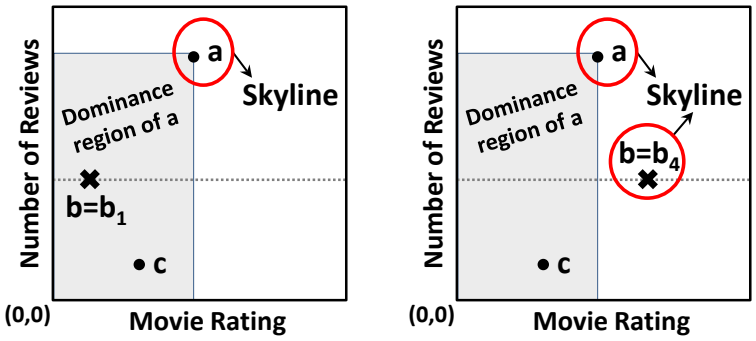


Figure 5.1: An incomplete dataset with three tuples, $\{a, b, c\}$: the incomplete tuple b with unknown "Movie Rating" (represented by a dotted line) may correspond to any of the different possible complete tuples, $b_1, b_2, b_3, b_4, \ldots$



(a) If the complete version of b is given by b_1, then a is the only result in the skyline

(b) If the the complete version of b is given by b_4, then b is in the skyline as well

Figure 5.2: The skyline of the incomplete dataset shown in Figure 5.1 can change based on the numerous complete possibilities of tuple b

5.1.2 Incomplete Data and Skylines

Missing values can complicate the definition of skylines and lead to extra overheads in skyline processing [2]. Khalefa *et. al* were among the first teams to propose algorithms for skyline queries over incomplete datasets [34]. They provided a new definition of dominance relation for incomplete data. Intuitively, their proposed definition is equivalent to applying the traditional dominance check only to the known common attributes of the pair of tuples that are being compared.

While being simple and (at the first glance) intuitive, this definition of domi-

nance leads to undesirable artifacts and counter-intuitive results. These include *non-transitive* and *cyclic* dominance, both of which are incompatible with any intuitive interpretation of skylines (e.g. cyclic dominance can lead to a scenario in which the skyline set is empty) and are also detrimental to efficient skyline processing (e.g. loss of transitivity renders useless many of the existing optimization techniques like indexing and data pruning). An example illustrating potentially counter-intuitive consequences of the above definition is given in Figure 5.2: here we consider the case where the unknown movie rating in Figure 5.1 becomes known at a later time. As we see in this figure, depending on the value the previously unknown movie rating of b takes, the skyline may contain only a or both a and b. Unfortunately, the dominance definition in [34] will simply return a as the skyline based on the common known attribute *Number of Reviews* of a and b, and thus, will ignore the *potential* of b being in the skyline.

Another way to process skyline queries over incomplete data is to reformulate them as probabilistic/set skyline queries [57, 3, 45, 35, 1, 43], where each NULL can be replaced with any value in its domain (with some non-zero probability). However, this will lead to an explosion in skyline evaluation costs as many of the optimizations applicable when the potential values of a tuple are clustered will not be applicable.

5.1.3 Main Contributions

Motivated by the fact that traditional dominance applied to incomplete data leads to *non-transitive* and *cyclic* dominance relationships, both of which are incompatible with any intuitive interpretation of skylines, this research work aims at developing intuitive and efficient ways of handling skyline queries over incomplete data sources. The main contributions detailed in this chapter are as follows:

- We propose two new definitions of dominance to help identify *potential domi-*

nance relations between tuples.

- Relying on these definitions, we introduce a novel skyline operator, Strata-Skyline (SS), for incomplete data; this operator stratifies the tuples into *strata* or *layers* of varying degrees of, so called, *skyline potential*.

- We propose two ways of indexing incomplete data sources using bitmap indices, namely Explicit-Bitmaps-for-Unknowns (EBU) and Implicit-Bitmaps-for-Unknowns (IBU); bitmap indices are stored in a compressed form and bitwise logical operations are implemented in the compressed domain.

- We develop two efficient algorithms, namely Strata-Skyline-using-Explicit-Bitmaps-for-Unknowns (SS-EBU) and Strata-Skyline-using-Implicit-Bitmaps-for-Unknowns (SS-IBU), that are leveraged by the SS operator to carry out the stratification process.

- Lastly, we introduce a reuse technique that helps SS-EBU and SS-IBU reuse sub-results of bitwise operations when tuples have overlaps in attribute values.

The rest of the chapter is structured as follows: Section 5.2 presents the preliminaries and states the problem addressed in this chapter. In Section 5.3, we discuss the proposed algorithms in detail. Section 5.5 analyzes the time complexity of the algorithms. Section 5.6 introduces further optimizations. Section 5.7 outlines possible extensions of the proposed algorithms. Experimental evaluations, reported in Section 5.8, confirm the advantages of the Strata-Skyline operator.

5.2 Key Concepts

This section introduces the key concepts and provides preliminary formalisms we will utilize in the rest of the chapter. In particular, we present (a) two new

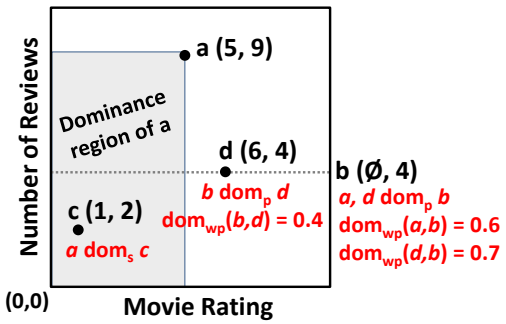


Figure 5.3: Tuples a and d *potentially-dominate* incomplete tuple b, tuple b *potentially-dominates* tuple d, whereas tuple a *strictly-dominates* tuple c

definitions of dominance that help identify *potential dominance relations* between tuples in incomplete data sources and (b) the concept of *strata* or *layers* of varying degrees of *skyline potential*. We also formally introduce (c) the Strata-Skyline (SS) operator for the efficient processing of skyline queries over incomplete data.

Throughout the chapter, we use the symbol "$\varnothing$" to represent missing/unknown values (i.e., NULL values). For example, the incomplete tuple b in Figure 5.3 is represented as $(\varnothing, 4)$, where $\varnothing$ indicates the missing value in attribute *Movie Rating* and 4 represents the know attribute *Number of Reviews*. Also, we assume that MAX is specified as the skyline preference function in which greater values are considered to be better.

5.2.1 Strict vs. Potential Dominance

According to the conventional definition [13] of dominance, a tuple t *dominates* another tuple q if only if tuple t is better than or equal to ($\geq$) tuple q in all attributes and better than ($>$) q in at least one attribute of the skyline attribute set. The skyline of a dataset, $\mathcal{D}$, then consists of the subset of tuples that are not dominated by any other tuple in $\mathcal{D}$. Khalefa *et al.* [34] adapt this definition to tuples with NULL values by focusing only on attributes for which both tuples have known values. In this work, we refer to this as *strict* dominance.

Definition 6 (Strict Dominance, dom_s)**.** *Let u and t be two tuples in $\mathcal{D}$ and $t.a_h$ be the value of attribute a_h of tuple t. Let A_S be the skyline attribute set and $A'_S \subseteq A_S$ be the subset for which both u and t have known values. Tuple u strictly-dominates t (u dom_s t) in the attribute set A_S iff*

$$(\forall a_i \in A'_S \;\; u.a_i \geq t.a_i) \wedge (\exists a_k \in A'_S \;\; u.a_k > t.a_k).$$

$\diamond$

For example, in Figure 5.3, tuple a *strictly-dominates* (dom_s) b since tuple $a(5, 9)$ is better than tuple $b(\varnothing, 4)$ along the attribute for which both tuples have known values ($9 > 4$).

5.2.2 (Unweighted) Potential Dominance

As we see in Figure 5.2, a tuple with NULL values has the *potential* of being a skyline even if it is dominated by the others on all known attributes. Therefore, in the work, we first propose an alternative dominance relationship in the presence of NULL values, which slightly relaxes the underlying constraint to take this into account.

Definition 7 ((Unweighted) Potential Dominance, dom_p)**.** *Let u, t be two tuples in $\mathcal{D}$. Let A_S be the skyline attribute set. Tuple u* potentially-dominates *tuple t (u dom_p t) in A_S iff tuple u is better than or equal to ($\geq$) tuple t in all common attributes that are known in both u and t:*

$$(\exists a_k \in A_S \; (u.a_k = \varnothing \vee t.a_k = \varnothing)) \; \wedge$$
$$(\forall a_i \in A_S \;\; (u.a_i \neq \varnothing \wedge t.a_i \neq \varnothing) \rightarrow (u.a_i \geq t.a_i)) \, .$$

$\diamond$

Intuitively, this implies that tuple u **may or may not** dominate tuple t *if and when the missing attribute values are identified.* Note that the potential dominance (dom_p)

relationship is not defined when neither tuple has a NULL value – in this case strict dominance (dom_s) needs to be sought.

Example 9. *In Figure 5.3, tuple a potentially-dominates (dom_p) b because b has an unknown value in the attribute* Movie Rating *and* $a(5, 9)$ *is better than or equal to ($\geq$)* $b(\varnothing, 4)$ *in the know common attribute* Number of Reviews. *This means that, unlike in the case of strict dominance [34], if the unknown value for b's* Movie Rating *is discovered to be* > 5*, then a will not dominate b. Thus, a should not eliminate tuple b from consideration as there is a* potential *for b to be in the skyline.*

Note that strict dominance treats pairs of tuples b/d and a/d similarly – i.e., there is no dominance relationship. Instead, potential dominance differentiates these pairs: b and d potentially-dominate each other, whereas a and d do not. ◦

5.2.3 Domain Weighted Potential Dominance

One weakness of the above definition of potential dominance is that it does not take in to account the domains of the unknown values. We address this by introducing a *domain weighted potential dominance*, defined as follows:

Definition 8 (Domain Weighted Potential Dominance, dom_{wp})**.** *Let u, t be two tuples in $\mathcal{D}$. Let A_S be the skyline attribute set. Tuple u potentially-dominates tuple t in the skyline attribute set A_S* with a weight $dom_{wp}(u, t)$ *iff*

$$dom_{wp}(u, t) = \frac{\|\langle u_c, t_c\rangle \text{ such that } (u_c \; dom_s \; t_c)\|}{\|\langle u_c, t_c\rangle\|},$$

where u_c and t_c are possible completions of tuples u and t. ◇

Note that $dom_{wp}(u, t)$ can be computed in constant time:

$$dom_{wp}(u, t) = \prod_{a_i \in A_S} \omega(u.a_i, t.a_i),$$

where, under the discrete domain distribution assumption,

- if $u.a_i \neq \varnothing$ and if $t.a_i \neq \varnothing$, then

$$\omega(u.a_i, t.a_i) = [u.a_i \geq t.a_i] \text{ (\textit{i.e.} 1 if } u.a_i \geq t.a_i \text{ and 0 otherwise)},$$

- if $u.a_i \neq \varnothing$ and if $t.a_i = \varnothing$, then

$$\omega(u.a_i, t.a_i) = \frac{\|domain_{\leq u.a_i}(a_i)\|}{\|domain(a_i)\|},$$

- if $u.a_i = \varnothing$ and if $t.a_i \neq \varnothing$, then

$$\omega(u.a_i, t.a_i) = \frac{\|domain(a_i)\| - \|domain_{<t.a_i}(a_i)\|}{\|domain(a_i)\|},$$

- and if $u.a_i = t.a_i = \varnothing$, then

$$\omega(u.a_i, t.a_i) = 0.5,$$

where $domain(a_i)$ is the domain of the attribute $a_i \in A_S$, $domain_{\leq X}(a_i)$ is the set of the values in the domain of a_i that are less than or equal to X, and $domain_{<X}(a_i)$ is the set of the values in the domain of a_i that are strictly less than X.

If the domain distribution is assumed to be continuous and independent across each skyline attribute, then $\omega(u.a_i, t.a_i)$ is defined as follows:

- if $u.a_i \neq \varnothing$ and if $t.a_i \neq \varnothing$, then, as before,

$$\omega(u.a_i, t.a_i) = [u.a_i \geq t.a_i] \text{ (\textit{i.e.} 1 if } u.a_i \geq t.a_i \text{ and 0 otherwise)},$$

- if $u.a_i \neq \varnothing$ and if $t.a_i = \varnothing$, then

$$\omega(u.a_i, t.a_i) = \frac{\int_{domain_{min}(a_i)}^{u.a_i} f(x)\,\mathrm{d}x}{\int_{domain_{min}(a_i)}^{domain_{max}(a_i)} f(x)\,\mathrm{d}x},$$

- if $u.a_i = \varnothing$ and if $t.a_i \neq \varnothing$, then

$$\omega(u.a_i, t.a_i) = \frac{\int_{domain_{min}(a_i)}^{domain_{max}(a_i)} f(x)\,\mathrm{d}x - \int_{domain_{min}(a_i)}^{t.a_i} f(x)\,\mathrm{d}x}{\int_{domain_{min}(a_i)}^{domain_{max}(a_i)} f(x)\,\mathrm{d}x},$$

- and if $u.a_i = t.a_i = \varnothing$, then, as described earlier,

$$\omega(u.a_i, t.a_i) = 0.5,$$

where $f(x)$ is a function that gives the distribution of the values in the domain of the attribute $a_i \in A_S$, $domain_{min}(a_i)$ is the minimum value in the domain of attribute a_i, and $domain_{max}(a_i)$ is the maximum value in the domain of a_i.

In the rest of this chapter, the distribution of the values in the domains of the skyline attributes is assumed to be discrete and uniform.

Example 10. *Assuming that the movie ratings uniformly take values from* 0 *to* 9*, the tuple* $a(5, 9)$ *in Figure 5.3 potentially dominates* $b(\varnothing, 4)$ *with weight*

$$dom_{wp}(a, b) = \frac{(5+1)}{(9+1)} \times 1 = 0.6.$$

The tuple $b(\varnothing, 4)$*, on the other hand, potentially-dominates tuple* $d(6, 4)$ *with weight*

$$dom_{wp}(b, d) = \frac{(9+1) - (5+1)}{(9+1)} \times 1 = 0.4.$$

∘

5.2.4 Strata-Skyline (SS) Queries

Given (1) a NULL-valued dataset, $\mathcal{D}(a_1, a_2, \ldots, a_d)$, and (2) a set of skyline attributes, $A_S \subseteq \{a_1, a_2, \ldots, a_d\}$, a Strata-Skyline query seeks to differentiate among tuples based on their *potential*s to be in the skyline. The *skyline potential* of a tuple is defined as follows:

Definition 9 ((Unweighted) Skyline Potential, $sp_u()$)**.** *Let* t *be a tuple in* $\mathcal{D}$*. Let* A_S *be the skyline attribute set. We say that the* unweighted skyline potential of tuple t *(*$sp_u(t)$*) is inversely proportional to the number of tuples that potentially-dominate* t

in the given dataset, that is:

$$sp_u(t) \sim \frac{1}{\mathcal{N}(t)},$$

where

$$\mathcal{N}(t) = \|\{u \in \mathcal{D} \text{ such that } (u \neq t) \wedge (u \; dom_p \; t)\}\|$$

is the number of tuples in $\mathcal{D}$ that potentially-dominate tuple t. ◇

Definition 10 (Domain Weighted Skyline Potential, $sp_w()$)**.** *Let t be a tuple in $\mathcal{D}$. Let A_S be the skyline attribute set. We define the* domain weighted skyline potential of tuple t *($sp_w(t)$) as:*

$$sp_w(t) \sim \frac{1}{\mathcal{R}(t)},$$

where

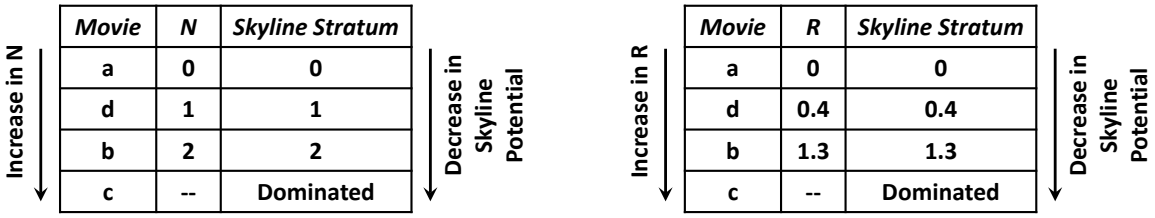
$$\mathcal{R}(t) = \sum_{(u \in \mathcal{D}) \wedge (u \neq t)} dom_{wp}(u, t).$$

◇

More specifically, a Strata-Skyline (SS) query *stratifies/groups* the tuples that are not *strictly-dominated* by any other tuple in a dataset, $\mathcal{D}$, into a set of *skyline strata (or layers)*, $\mathcal{S} = \{S_0, S_1, S_2, \dots\}$. The skyline strata are arranged in the descending order of the skyline potentials. A skyline stratum (or layer) is defined as follows:

Definition 11. *(Skyline Stratum, S_i) A* skyline stratum*, S_i, is a set of tuples $\{t_1, t_2, t_3, \dots\}$ such that $sp_u(t_1) = sp_u(t_2) = sp_u(t_3) = \dots$, where $sp_u(t_i)$ represents the unweighted skyline potential of tuple t_i. Similarly, if domain weighted skyline potential is used, then a* skyline stratum*, S_i, is a set of tuples $\{t_1, t_2, t_3 \dots\}$ such that $sp_w(t_1) = sp_w(t_2) = sp_w(t_3) = \dots$, where $sp_w(t_i)$ is the unweighted skyline potential of tuple t_i.* ◇

Example 11 (Strata-Skyline (SS) Query)**.** *Given a movies dataset,* Movies (movieID, rating, numberOfReviews)*, containing incomplete information about movie ratings, the query:*



(a) Obtained using unweighted potential dominance (dom_p)

(b) Obtained using domain weighted potential dominance (dom_{wp})

Figure 5.4: The set of skyline strata produced by the Strata-Skyline operator on the example movie dataset in Figure 5.3

```
Skyline Strata = SS * FROM Movies
                     WHERE rating MAX,
                     numberOfReviews MAX
```

would return a set of skyline strata of movies based on the skyline attributes rating *and* numberOfReviews.

Figure 5.4(a) shows the skyline strata for the example data in Figure 5.3 assuming unweighted skyline potentials. Movie a is in S_0 *as it is not potentially-dominated by any other tuples, whereas movie c is in the lowest stratum and is pruned away because it is dominated in the traditional sense by a. Movie b, on the other hand, is not pruned but forms stratum* S_2 *since tuples a and d potentially-dominate b. Stratum* S_1 *is above* S_2 *and contains movie d since it is potentially-dominated only by tuple b.*

Figure 5.4(b) shows the skyline strata obtained using the domain weighted potential dominance definition. Movie a, as before, is in S_0 *as it is not potentially-dominated by any other tuples in the dataset. On the other hand, movie d is placed in* $S_{0.4}$ *and movie b is in* $S_{1.3}$*, which indicate that tuples d and b have higher weighted skyline potentials as compared to their corresponding unweighted skyline potentials*[1]*.* ∘

[1]In this example, the skyline strata did not change as a result of domain weighting. However, in general, domain weighting can impact skyline strata.

Naturally, the user may be interested in (and thus shown) only the top few skyline strata with the highest potentials.

5.2.5 *Resolving Cycles and Non-Transitivity with Stratification*

It is important to note that the definition of potential dominance is not-cycle free. For example, for tuples $u(5, 6, \varnothing)$, $v(\varnothing, 3, 2)$, and $w(7, \varnothing, 1)$, the following cyclic relationship holds:

$$u \; dom_p \; v \; dom_p \; w \; dom_p \; u.$$

Similarly, potential dominance may be non-transitive; for instance, given $x(5, 6, \varnothing)$, $y(\varnothing, 3, \varnothing)$, and $z(8, 1, \varnothing)$, we have

$$x \; dom_p \; y \; dom_p \; z,$$

but x does not potentially-dominate z.

Despite these, it is also easy to see that the *skyline potential* function introduces a (transitive and acyclic) partial-order on the tuples in a dataset (i.e., their skyline stratum numbers), thereby not suffering from the undesirable artifacts of the existing skyline operators, such as [34], which are based on the strict definition of dominance.

5.3 Strata-Skyline (SS) Query Processing based on Unweighted Potential Dominance

In this section, we present two Strata-Skyline (SS) algorithms, namely Strata-Skyline-using-Explicit-Bitmaps-for-Unknowns (SS-EBU) and Strata-Skyline-using-Implicit-Bitmaps-for-Unknowns (SS-IBU), to efficiently stratify the tuples in a NULL-valued data source. These algorithms consider the unweighted potential dominance definition presented in Section 5.2.2. In Section 5.4, we discuss the extension of these algorithms to the domain weighed potential dominance defined in Section 5.2.3.

Input		
a_1	a_2	a_3
1	1	2
3	0	1
1	1	0
1	0	0

a_1	
3 (B_{11})	1 (B_{12})
0	1
1	1
0	1
0	1

a_2	
1 (B_{21})	0 (B_{22})
1	1
0	1
1	1
0	1

a_3		
2 (B_{31})	1 (B_{32})	0 (B_{33})
1	1	1
0	1	1
0	0	1
0	0	1

Figure 5.5: The bitmap structure proposed by Tan *et al.* for processing skyline queries over data sources with no missing values [70]

5.3.1 The SS-EBU Algorithm

The `SS-EBU` algorithm uses an extended version of the bitmap structure proposed in [70] for progressive skyline processing on complete datasets. In particular, `SS-EBU` carries out the stratification process using fast bitwise operations over compressed bitmap structures [82, 66, 40] as opposed to expensive tuple-to-tuple comparisons. In this section, we first elucidate the use of bitmaps for traditional skyline computation [70], next introduce the proposed Explicit-Bitmaps-for-Unknowns (`EBU`) structure, and then, present the details underlying the novel `SS-EBU` algorithm.

Bitmap Encoding for Complete Data and Bitmap-based Skyline Processing

[70] assumes that an attribute i ($1 \leq i \leq d$) in tuple $t = (t_1, t_2, \ldots, t_d)$ has k_i distinct values and p_{ij} denotes the jth distinct value of the ith attribute, where $p_{i1} > p_{i2} > \cdots > p_{ik_i}$. A tuple t is encoded as an m-bit vector in which t_i is represented by k_i bits. Hence, the size of the bit vector is $m = \sum_{i=1}^{d} k_i$. Let the jth bit in m correspond to p_{ij}. For the MAX skyline preference function, the first bit in m corresponds to p_{i1}, which in turn represents the largest value in attribute i, the second bit in m corresponds to p_{i2} that relates to the second largest value in attribute i, and so on. If t_i is the p_{iq}th distinct value of attribute i, then the k_i bits representing t_i are set such that bits from 1 to $q-1$ are set to *0*, while bits from q

to k_i are set to *1*. Intuitively, *this encodes the "$\geq$" and "$<$" relationships in a binary form, where* 1 *encodes "$\geq$" and* 0 *encodes "$<$".*

Example 12. *Figure 5.5 shows an example dataset that contains four tuples each with three attributes. The first attribute (a_1) has two distinct values (3, 1), the second attribute (a_2) again has two distinct values (1, 0) and the third attribute (a_3) has three distinct values (2, 1, 0). Let us consider the second tuple* $[3, 0, 1]$. *The value in its third attribute (a_3) is 1, which is the second largest value in that attribute. So, only the bit corresponding to 2 (bitmap B_{31}) is set to 0, while the other bitmaps (B_{32} and B_{33}) are set to 1, resulting in the sequence* 011 *in the bit representation of the third attribute (a_3). Similarly, the second attribute (a_2) of* $[3, 0, 1]$ *has the value 0 and so only the bits corresponding to values larger than 0 (in this case, only bitmap B_{21} that corresponds to value 1) will be set to 0, while the rest (only bitmap B_{22} in this case) are set to 1. This leads to the sequence 01 in the second attribute (a_2) for the tuple* $[3, 0, 1]$. *Finally, using the same logic, the bit sequence corresponding to the first attribute (a_1) of the tuple* $[3, 0, 1]$ *is 11.* ∘

Let B_{ij} denote the bitmap for the jth distinct value of the ith attribute. To check if a tuple $t = (t_1, t_2, \ldots, t_d)$ is in the skyline, [70] performs the following logical bitwise AND/OR operations:

1. $A = B_{1q_1}\ AND\ B_{2q_2}\ AND\ \ldots\ AND\ B_{dq_d}$, where AND represents the bitwise AND operation, and t_i is the q_ith distinct value of attribute i. Bit-slice A has the property that the nth bit is set to 1 if and only if the nth tuple in the dataset has a value in each attribute that is greater than or equal to the value of the corresponding attribute in tuple t.

2. $O = B_{1q_1-1}\ OR\ B_{2q_2-1}\ OR\ \ldots\ OR\ B_{dq_d-1}$, where OR represents the bitwise OR operation. This OR operation is carried out on the preceding bitmap of

B_{iq_i}, corresponding to the smallest value that is larger than t_i. If there is no preceding bitmap for attribute i, i.e. if q_i corresponds to the first bitmap, B_{i0} is set to all 0s. Bit-slice O is such that the nth bit is set to 1 if and only if the nth tuple has some of its attribute values greater than the value of the corresponding attributes in t.

3. $D = A\ AND\ O$, where AND represents the bitwise AND operation. Bit-slice D is such that the nth bit is set to 1 if and only if the nth tuple has each of its attribute values greater than or equal to the corresponding attribute values in t and some of its attribute values are strictly greater than the corresponding attribute values in tuple t. Thus, if the nth bit is set to 1, it can be concluded that the nth tuple dominates (dom_s) tuple t.

Note that if the bit-slice D contains only zeros, it indicates that there are no tuples in the dataset that dominate (dom_s) t.

Example 13. *Let us consider the input dataset in Figure 5.5 and determine whether the tuple* $[3, 0, 1]$ *is in the skyline or not. First, the bitwise AND operation is carried out between the bitmaps* B_{11}*,* B_{22} *and* B_{32}*:*

$$A = 0100\ AND\ 1111\ AND\ 1100 = 0100$$

Next, we carry out the bitwise OR operation between the preceding bitmaps B_{10}*,* B_{21} *and* B_{31}*:*

$$O = 0000\ OR\ 1010\ OR\ 1000 = 1010$$

Finally, we carry out the bitwise AND operation between the bit-slices A *and* O*:*

$$D = 0100\ AND\ 1010 = 0000$$

Bit-slice D *contains only zeros, thus no tuple in the dataset dominates* $[3, 0, 1]$*. Hence, tuple* $[3, 0, 1]$ *is a skyline tuple.* ◦

Input			a_1			a_2			a_3			
a_1	a_2	a_3	3 (B_{11})	1 (B_{12})	Ø ($B_{1\emptyset}$)	1 (B_{21})	0 (B_{22})	Ø ($B_{2\emptyset}$)	2 (B_{31})	1 (B_{32})	0 (B_{33})	Ø ($B_{3\emptyset}$)
Ø	1	2	0	0	1	1	1	0	1	1	1	0
3	Ø	1	1	1	0	0	0	1	0	1	1	0
1	1	0	0	1	0	1	1	0	0	0	1	0
1	0	0	0	1	0	0	1	0	0	0	1	0

Figure 5.6: The Explicit-Bitmaps-for-Unknowns (EBU) structure: positions corresponding to unknown attribute values in the bitmaps for known values (B_{11}, B_{12}, B_{21}, B_{22}) are set to *zero* (highlighted above); bitmaps for unknown values ($B_{1\emptyset}$, $B_{2\emptyset}$, $B_{3\emptyset}$) are *explicitly* used during an SS operation

Explicit-Bitmaps-for-Unknowns (EBU) Index for Incomplete Data

Our proposed Explicit-Bitmaps-for-Unknowns (EBU) structure consists of two types of bitmap structures: (a) bitmaps for known attribute values and (b) bitmaps for unknown (or NULL) values. The construction of the two components in the EBU bitmap structure is described below.

• **Bitmap Construction for Known Attribute Values.** Let B_{ij} denote the bitmap for the jth distinct known value of the ith attribute. Bitmaps for known attribute values of an incomplete dataset are built using the bitmap encoding shown in Section 5.3.1, with the added constraint that the positions corresponding to *unknown* attribute values are set to *0*.

Example 14. *In Figure 5.6 we have an input dataset which contains four tuples, each with three attributes. The first attribute (a_1) has two distinct known values (3, 1) and one unknown value (∅ in tuple $[\emptyset, 1, 2]$), the second attribute (a_2) again has two distinct known attribute values (1, 0) and one unknown value (∅ in tuple $[3, \emptyset, 1]$), and the third attribute (a_3) has three distinct known values (2, 1, 0) and zero unknown values.*

Firstly, let us consider the second tuple $[3, \emptyset, 1]$. The value in its third attribute (a_3) is known and is equal to 1, which is the second largest value in that attribute.

So (as usual) only the bit corresponding to 2 (bitmap B_{31}) is set to 0, while the bits in other bitmaps (B_{32} and B_{33}) are set to 1, resulting in the sequence 011 *in the bit representation for known values of the third attribute (a_3). The second attribute (a_2) of $[3, \varnothing, 1]$, however, has a NULL value, represented by $\varnothing$. In this case, all corresponding bits in the bitmaps for known values of the second attribute (B_{21} and B_{22}) will be set to 0 (highlighted in Figure 5.6). This leads to the sequence* 00 *for tuple $[3, \varnothing, 1]$ in the bit representation for* known values *of the attribute a_2.* ◦

• **Bitmap Construction for the NULL Attribute Values.** Let $B_{i\varnothing}$ denote the bitmap for the unknown (NULL) values of the ith attribute. If t_i represents an unknown value of attribute i in tuple $t = (t_1, t_2, \ldots, t_d)$, then the corresponding bit in $B_{i\varnothing}$ is set to *1*, otherwise it is set to *0*.

Example 15. *In Figure 5.6, the second attribute (a_2) of tuple $[3, \varnothing, 1]$ has an unknown value. In this scenario, the corresponding bit in the bitmap for unknown values of the second attribute ($B_{2\varnothing}$) will be set to 1. On the other hand, the first attribute (a_1) of tuple $[3, \varnothing, 1]$ does not have a NULL value. In this case, the corresponding bit in the bitmap ($B_{1\varnothing}$) for unknown values of the first attribute is set to 0.* ◦

SS Query Processing using Explicit-Bitmaps-for-Unknowns (SS-EBU)

The proposed `SS-EBU` algorithm avoids expensive tuple-to-tuple comparisons by utilizing fast bitwise operations over the bitmaps in the `EBU` structure. Let $t = (t_1, t_2, \ldots, t_d)$ be a tuple in the dataset.

Case I. If tuple t is incomplete (i.e., t has some unknown (NULL) attribute values), then `SS-EBU` operates as follows:

1. The `SS-EBU` algorithm first computes the bit-slice $P = (B_{1\varnothing}\ OR\ B_{1q_1})\ AND\ (B_{2\varnothing}\ OR\ B_{2q_2})\ AND\ \ldots\ AND\ (B_{d\varnothing}\ OR\ B_{dq_d})$, for all $t_i \neq \varnothing$ that is q_ith

distinct value at attribute i. In other words, this operation does not include the unknown attributes in t.

Bit-slice P has the property that the nth bit is set to 1 if and only if the nth tuple in a dataset has value in each attribute that is either unknown or greater than or equal to the value of the corresponding attribute in tuple t. In other words, if the nth bit is set to 1, then this indicates that the nth tuple *potentially-dominates* tuple t. Therefore, P helps find the set of tuples that *potentially-dominate* tuple t.

2. Next, `SS-EBU` computes $\mathcal{N}(t) = cardinality(P) - 1$, where $\mathcal{N}(t)$ is the total number of tuples that *potentially-dominate* t and $cardinality(P)$ gives the count of number of bits in bit-slice P set to 1. $\mathcal{N}(t)$ is one less than cardinality of P to avoid considering *potential-dominance* of t by itself.

3. Finally, `SS-EBU` inserts tuple t into the appropriate *skyline stratum* (or *layer*) $S_j \in \mathcal{S}$ based on $\mathcal{N}(t)$.

Case II. If tuple t has no NULL values, i.e. t is complete, then the `SS-EBU` algorithm operates differently:

1. The algorithm computes bit-slice $D = A\ AND\ O$, where A and O are computed as in Section 5.3.1: if the nth bit in D is set to 1, then this indicates that the nth tuple *strictly-dominates* tuple t.

2. Next the algorithm computes $cardinality(D)$, where $cardinality(D)$ gives the count of the number of bits in bit-slice D that are set to 1.

3. If $cardinality(D) = 0$, i.e. there are no tuples in the dataset that *strictly-dominate* tuple t:

(a) SS-EBU computes bit-slice P as in Case I, Step 1.

(b) It calculates $\mathcal{N}(t)$ as in Case I, Step 2.

(c) It inserts tuple t into the appropriate *skyline stratum* $S_j \in \mathcal{S}$ based on $\mathcal{N}(t)$ as in Case I, Step 3.

4. If $cardinality(D) > 0$, however, at least one tuple in the dataset *strictly-dominates* t. Thus, t is not inserted into the *skyline strata* set $\mathcal{S}$.

A detailed pseudocode of SS-EBU is presented in Figure 5.7.

Example 16. *Let us consider the input in Figure 5.6 and determine how* $[\varnothing, 1, 2]$ *is inserted into* $\mathcal{S}$*:* $[\varnothing, 1, 2]$ *contains a NULL value in the 1st attribute* (a_1)*. Hence, we compute* P *by performing bitwise OR/AND operations only on the known values (1, 2) in the 2nd* (a_2) *and 3rd* (a_2) *attributes:*

$$\begin{aligned} P &= (B_{2\varnothing}\ OR\ B_{21})\ AND\ (B_{3\varnothing}\ OR\ B_{31}) \\ &= (0100\ OR\ 1010)\ AND\ (0000\ OR\ 1000) \\ &= 1110\ AND\ 1000 = 1000. \end{aligned}$$

Next, we compute $\mathcal{N}([\varnothing, 1, 2])$*:*

$$\mathcal{N}([\varnothing, 1, 2]) = cardinality(P) - 1 = 0.$$

Consequently, tuple $[\varnothing, 1, 2]$ *is inserted into the* skyline stratum S_0*. Figure 5.8 shows the complete* skyline strata *returned by the* SS-EBU *algorithm for the given input.* ∘

5.3.2 The SS-IBU Algorithm

Unlike SS-EBU, our second proposed algorithm, SS-IBU, does not use bitmaps for NULL (unknown) attribute values – instead it relies on the bitmap that represents

Algorithm 1: SS-EBU($\mathcal{D}, A_S$, EBU)

Input:

$\mathcal{D}$: An incomplete dataset, A_S: A set of skyline attributes

EBU: Explicit-Bitmaps-for-Unknowns index structure

Output:

$\mathcal{S}$: A set of *skyline strata*

Procedure:

for each tuple $t = (t_1, t_2, \ldots, t_d)$ in $\mathcal{D}$ **do**

if $t_1 = \varnothing \vee t_2 = \varnothing \vee \ldots \vee t_d = \varnothing$ **then**

P := (EBU.$B_{1\varnothing}$ OR EBU.B_{1q_1}) AND (EBU.$B_{2\varnothing}$ OR EBU.B_{2q_2}) AND ... AND (EBU.$B_{d\varnothing}$ OR EBU.B_{dq_d})

$\mathcal{N}(t) := cardinality(P) - 1$

Insert tuple t into the appropriate *skyline stratum* S_j in $\mathcal{S}$ based on $\mathcal{N}(t)$

end if

if $t_1 \neq \varnothing \wedge t_2 \neq \varnothing \wedge \ldots \wedge t_d \neq \varnothing$ **then**

$A :=$ EBU.B_{1q_1} AND ... AND EBU.B_{dq_d}

$O :=$ EBU.B_{1q_1-1} OR ... OR EBU.B_{dq_d-1}

$D := A$ AND O

if $cardinality(D) = 0$ **then**

P := (EBU.$B_{1\varnothing}$ OR EBU.B_{1q_1}) AND (EBU.$B_{2\varnothing}$ OR EBU.B_{2q_2}) AND ... AND (EBU.$B_{d\varnothing}$ OR EBU.B_{dq_d})

$\mathcal{N}(t) := cardinality(P) - 1$

Insert t into the appropriate *skyline stratum* S_j in $\mathcal{S}$ based on $\mathcal{N}(t)$

else

Tuple t is *strictly-dominated* and it is not inserted into $\mathcal{S}$

end if

end if

end for

return $\mathcal{S}$

Figure 5.7: The SS-EBU algorithm

the set of tuples with no missing values during query processing. This section first presents the Implicit-Bitmaps-for-Unknowns (IBU) structure, and then, explains the details behind the SS-IBU algorithm.

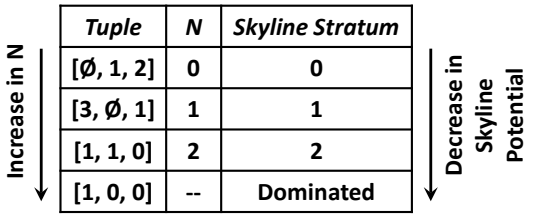


Figure 5.8: An example result of an SS query based on unweighted potential dominance (dom_p) obtained via the SS-EBU and SS-IBU algorithms

Implicit-Bitmaps-for-Unknowns (IBU) Index for Incomplete Data

The proposed IBU structure is made up of two types of bitmap structures: (a) bitmaps for known attribute values and (b) a bitmap that represents the set of tuples with no missing/unknown attribute values. The construction of the two components in the IBU bitmap structure is described below.

• **Bitmap Construction for Known Attribute Values.** Let B_{ij} denote the bitmap for the jth distinct known value of the ith attribute. Similar to the EBU structure (Section 5.3.1), the bitmaps for the known attribute values of an incomplete dataset in the IBU structure are also constructed using the bitmap encoding described in Section 5.3.1, except that the positions corresponding to *unknown* attribute values are set to *1* instead of *0*, to *implicitly* represent missing values, thus the name Implicit-Bitmaps-for-Unknowns.

Example 17. *Figure 5.9 shows an example. Once again, we have a dataset that contains four tuples each with three attributes. Let us again consider the second tuple* $[3, \varnothing, 1]$. *As explained before in Section 5.3.1, only the bit corresponding to 2 (bitmap* B_{31}*) is set to 0, while the bits in other bitmaps (*B_{32} *and* B_{33}*) are set to 1, resulting in the sequence* 011 *in the bit representation for known values of the third attribute (*a_3*). Now let us look at an example of the deviation from the encoding presented before. The second attribute (*a_2*) of* $[3, \varnothing, 1]$ *has an unknown value. In this case, all corresponding bits in the bitmaps for known values of the second attribute (*B_{21} *and*

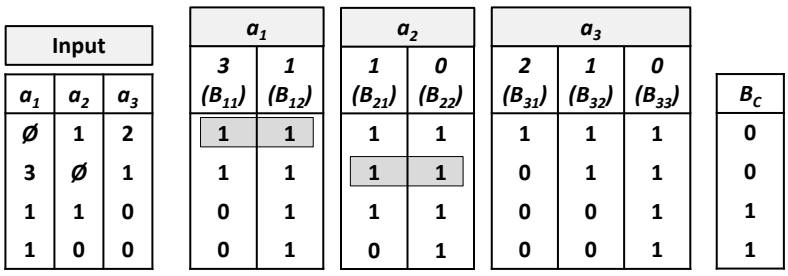


Figure 5.9: The Implicit-Bitmaps-for-Unknowns (IBU) structure: positions corresponding to unknown attribute values in the bitmaps for known values (B_{11}, B_{12}, B_{21}, B_{22}) are set to *1* (highlighted above) to *implicitly* indicate NULL values; a bitmap that represents the set of tuples with no missing values (B_C) is utilized during Strata-Skyline (SS) computation

B_{22}) are set to 1 (highlighted in Figure 5.9). This leads to the bit sequence 11 *for tuple* $[3, \varnothing, 1]$ *in the bit representation for known values of the second attribute (a_2).* ∘

• **Bitmap Representation of Tuples with No Unknown Values.** B_C is a bitmap that represents the set of all *complete* tuples (with no NULL values) in the data source.

Example 18. *Figure 5.9 illustrates this: in this example, B_C is 0011 as the first two tuples have NULL values, but the last two tuples do not have NULL values.* ∘

SS Query Processing using Implicit-Bitmaps-for-Unknowns (SS-IBU)

The SS-IBU algorithm performs the following steps on each tuple $t = (t_1, t_2, \ldots, t_d)$ in the dataset:

Case I. If tuple t has any unknown (NULL) attribute values, i.e. t is incomplete, then SS-IBU performs as follows:

1. The algorithm computes the bit-slice $P = B_{1q_1}\ AND\ B_{2q_2}\ AND\ \ldots\ AND\ B_{dq_d}$, where $t_i \neq \varnothing$ and it is the q_ith distinct value at attribute i. In other words, this operation does not include the unknown attributes in t. Bit-slice P has the property that the nth bit is set to 1 if and only if the nth tuple has value in

each attribute that is either unknown or greater than or equal to the value of the corresponding attribute in t. Thus, if the nth bit is set to 1, the nth tuple *potentially-dominates* tuple t.

It is important to note that the bitwise OR operations used by `SS-EBU` to compute P (Section 5.3.1, Case I, Step 1) are avoided by the `SS-IBU` algorithm.

2. Next, the algorithm computes the number, $\mathcal{N}(t)$, of tuples that *potentially-dominate* t as in the `SS-EBU` algorithm in Section 5.3.1, Case I, Step 2.

3. Finally, tuple t is inserted into the appropriate *skyline stratum* $S_i \in \mathcal{S}$ based on the value of $\mathcal{N}(t)$.

Case II. If, on the other hand, tuple t has no NULL values, i.e. t is complete, then `SS-IBU` performs the following:

1. `SS-IBU` computes bit-slice P as in Case I, Step 1.

2. Next, it computes $P_C = P\ AND\ B_C$, where B_C is the bitmap that represents the set of tuples with no missing values. P_C has the property that the nth bit is set to 1 if and only if the nth tuple has no missing attribute values and has value in each attribute that is greater than or equal to the value of the corresponding attribute in t. If the nth bit is set to 1, then this indicates that the nth tuple is complete and may *strictly-dominate* t.

3. `SS-IBU` then calculates the number, $\mathcal{C}(t)$, of tuples that have no missing values and may *strictly-dominate* tuple t as $\mathcal{C}(t) = cardinality(P_C) - 1$.

4. If $\mathcal{C}(t) = 0$, there are no complete tuples in the dataset that *strictly-dominate* t. Thus, `SS-IBU` can

(a) reuse already computed bit-slice P to calculate $\mathcal{N}(t)$ as shown in Case I, Step 2 and, then,

(b) insert tuple t into the appropriate *skyline stratum* $S_i \in \mathcal{S}$ based on $\mathcal{N}(t)$.

5. If $\mathcal{C}(t) > 0$, then there is a possibility that there is a complete tuple in the dataset which *strictly-dominates* t. Thus, the algorithm confirms the existence of such a complete tuple in the dataset by carrying out the following bitwise AND/OR operations:

   (a) It computes bit-slice $O_U = B_{1q_1-1}\ OR\ B_{2q_2-1}\ OR\ \ldots\ OR\ B_{dq_d-1}$. This OR operation is carried out on the preceding bitmap of B_{iq_i}, corresponding to the smallest value that is larger than t_i. If there is no preceding bitmap for attribute i, i.e. if q_i corresponds to the first bitmap, then bitmap B_{i0} is set to 0.

   O_U has the property that the nth bit is set to 1 if and only if the nth tuple has some of its attribute values that are either unknown or greater than the value of the corresponding attribute in t.

   (b) `SS-IBU` then computes bit-slice $O_C = O_U\ AND\ B_C$, where B_C denotes the bitmap that represents the set of tuples with no NULL values. Bit-slice O_C has the property that the nth bit is set to 1 if and only if the nth tuple has no missing values and has some of its attribute values greater than the corresponding attribute values in t.

   (c) Next, `SS-IBU` combines P_C computed in Case II, Step 2 with O_C to obtain $D = P_C\ AND\ O_C$.

   Note that, if the nth bit in bit-slice D is set to 1, then this indicates that the nth tuple is complete and *strictly-dominates* tuple t.

(d) If $cardinality(D) = 0$, i.e. there are no complete tuples that *strictly-dominate* t, then `SS-IBU`

i. reuses already computed bit-slice P to calculate $\mathcal{N}(t)$ as shown in Case I, Step 2, then,

ii. inserts tuple t into the appropriate *skyline stratum* $S_i \in \mathcal{S}$ based on the value of $\mathcal{N}(t)$.

(e) If $cardinality(D) > 0$, then there is at least 1 complete tuple that *strictly-dominates* t. Thus, t goes to the lowest strata that is pruned away.

A detailed pseudocode of `SS-IBU` is presented in Figure 5.10.

Example 19. *Let us consider the input shown in Figure 5.9 and determine how the tuple* $[1, 1, 0]$ *is inserted into* $\mathcal{S}$*. It can be observed that tuple* $[1, 1, 0]$ *has no missing values. Hence, we first compute bit-slice* P *by performing the bitwise AND operation between the bitmaps* B_{12}*,* B_{21} *and* B_{33}*:*

$$P = 1111\ AND\ 1110\ AND\ 1111 = 1110.$$

Then, we compute bit-slice P_C*:*

$$P_C = P\ AND\ B_C = 1110\ AND\ 0011 = 0010.$$

Next, we compute $\mathcal{C}([1, 1, 0])$*:*

$$\mathcal{C}([1, 1, 0]) = cardinality(P_C) - 1 = 0.$$

Since $\mathcal{C}([1, 1, 0]) = 0$*, we compute* $\mathcal{N}([1, 1, 0])$ *using* P*:*

$$\mathcal{N}([1, 1, 0]) = cardinality(P) - 1 = 2.$$

Thus, as shown in Figure 5.8, tuple $[1, 1, 0]$ *is inserted into the* skyline stratum S_2*. Figure 5.8 also shows the complete set of* skyline strata *returned by* `SS-IBU` *for the given input dataset.* ◦

Algorithm 2: SS-IBU($\mathcal{D}, A_S$, IBU)

Input:

$\mathcal{D}$: An incomplete dataset, A_S: A set of skyline attributes

IBU: Implicit-Bitmaps-for-Unknowns index structure

Output:

$\mathcal{S}$: A set of *skyline strata*

Procedure:

for each tuple $t = (t_1, t_2, \ldots, t_d)$ in $\mathcal{D}$ **do**

if $t_1 = \varnothing \vee t_2 = \varnothing \vee \ldots \vee t_d = \varnothing$ **then**

$P :=$ IBU.B_{1q_1} AND IBU.B_{2q_2} AND $\ldots$ AND IBU.B_{dq_d}

$\mathcal{N}(t) := cardinality(P) - 1$

Insert tuple t into the appropriate *skyline stratum* S_i in $\mathcal{S}$ based on $\mathcal{N}(t)$

end if

if $t_1 \neq \varnothing \wedge t_2 \neq \varnothing \wedge \ldots \wedge t_d \neq \varnothing$ **then**

$P :=$ IBU.B_{1q_1} AND IBU.B_{2q_2} AND $\ldots$ AND IBU.B_{dq_d}

$P_C := P$ AND IBU.B_C

$\mathcal{C}(t) := cardinality(P_C) - 1$

if $\mathcal{C}(t) = 0$ **then**

$\mathcal{N}(t) := cardinality(P) - 1$

Insert t into the appropriate *skyline stratum* S_i in $\mathcal{S}$ based on $\mathcal{N}(t)$

else

$O_U :=$ IBU.B_{1q_1-1} OR IBU.B_{2q_2-1} OR $\ldots$ OR IBU.B_{dq_d-1}

$O_C := O_U$ AND IBU.B_C

$D := P_C$ AND O_C

if $cardinality(D) = 0$ **then**

$\mathcal{N}(t) := cardinality(P) - 1$

Insert t into the appropriate *skyline stratum* S_i in $\mathcal{S}$ based on $\mathcal{N}(t)$

else

Tuple t is *strictly-dominated* and it is not inserted into $\mathcal{S}$

end if

end if

end if

end for

return $\mathcal{S}$

Figure 5.10: The SS-IBU algorithm

5.4 The SS-WB Algorithm: Strata-Skyline (SS) Query Processing based on Domain Weighted Potential Dominance

The SS-EBU and SS-IBU algorithms presented in Section 5.3 consider the unweighted potential dominance definition presented in Section 5.2.2, and thus, leverage index structures that are encoded in binary form (1 encoding "≥" and 0 encoding "<") and logical bitwise AND/OR operations. These index structures and algorithms can be extended to answer Strata-Skyline (SS) queries based on the more general domain weighted potential (Definitions 8 and 10).

Here, we present the Strata-Skyline-using-Weighted-Bitmaps (SS-WB) algorithm that can be leveraged to efficiently stratify tuples in a Null-valued data source based on the domain weighed potential dominance definition (Section 5.2.3). Unlike SS-EBU and SS-IBU, the SS-WB algorithm does not use bitmaps that leverage a binary encoding of the ≥ relationship. Instead, SS-WB relies on domain weighted bitmaps whose entries are encoded using the $\omega(X, Y)$ values defined in Section 5.2.3. In this section, we first present the Weighted-Bitmaps (WB) structure, and then, explain the details behind the SS-WB algorithm.

5.4.1 Weighted-Bitmaps (WB) Index for Incomplete Data

The proposed WB structure is made up of three types of bitmap structures: (a) domain weighted bitmaps for known attribute values, (b) domain weighted bitmaps for unknown dimension values, and (c) a bitmap that represents the set of tuples with no missing/unknown attribute values. The construction of the three components in the WB bitmap structure described below.

• **Domain Weighted Bitmap Construction for Known Attribute Values.** Let B_{ij} denote the domain weighted bitmap for the jth distinct known value of the ith

Input			a_1			a_2			a_3				B_C
a_1	a_2	a_3	3 (B_{11})	1 (B_{12})	Ø ($B_{1\emptyset}$)	1 (B_{21})	0 (B_{22})	Ø ($B_{2\emptyset}$)	2 (B_{31})	1 (B_{32})	0 (B_{33})	Ø ($B_{3\emptyset}$)	
Ø	1	2	$\frac{5-3}{5}$	$\frac{5-1}{5}$	0.5	1	1	$\frac{2}{5}$	1	1	1	$\frac{3}{5}$	0
3	Ø	1	1	1	$\frac{4}{5}$	$\frac{5-1}{5}$	$\frac{5-0}{5}$	0.5	0	1	1	$\frac{2}{5}$	0
1	1	0	0	1	$\frac{2}{5}$	1	1	$\frac{2}{5}$	0	0	1	$\frac{1}{5}$	1
1	0	0	0	1	$\frac{2}{5}$	0	1	$\frac{1}{5}$	0	0	1	$\frac{1}{5}$	1

Figure 5.11: The Weighted-Bitmaps (WB) structure: each entry in the bitmaps for known and unknown values is encoded using the $\omega(X, Y)$ function defined in Section 5.2.3, assuming that the domains of the attributes are between $\{0, 1, 2, 3, 4\}$; the weighted bitmaps and a bitmap that represents the set of tuples with no missing values (B_C) are utilized during an SS operation based on domain weighted potential

attribute, $domain(i)$ be the domain of attribute i, and $domain_{<Y}(i)$ represent the set of values in the domain of i that are strictly less than Y, where $Y \neq \varnothing$. The domain weighted bitmaps for the known attribute values of an incomplete dataset in the WB bitmap structure are constructed using two of the $\omega(X, Y)$ encodings described in Section 5.2.2: (1) if $X \neq \varnothing$, then

$$\omega(X, Y) = [X \geq Y] \text{ (}i.e.\text{ 1 if } X \geq Y \text{ and 0 otherwise);}$$

this is similar to the binary encoding used by the EBU and IBU structures (Section 5.3), and (2) if $X = \varnothing$ then,

$$\omega(X, Y) = \frac{\|domain(i)\| - \|domain_{<Y}(i)\|}{\|domain(i)\|}.$$

Example 20. *Figure 5.11 shows an example. We have an input dataset that contains four tuples each with three attributes and the domains of the attributes are between* $\{0, 1, 2, 3, 4\}$. *The first attribute* (a_1) *has two distinct known values (3, 1) and one unknown value (*$\varnothing$ *in tuple* $[\varnothing, 1, 2]$*), the second attribute* (a_2) *again has two distinct known values (1, 0) and one unknown value (*$\varnothing$ *in tuple* $[3, \varnothing, 1]$*), and the third attribute* (a_3) *has three distinct known values (2, 1, 0) and zero unknown values.*

Let us consider the second tuple $[3, \varnothing, 1]$. *The bit corresponding to 2 (bitmap* B_{31}*) is set to* $\omega(1,2) = 0$, *while the bits in* B_{32} *and* B_{33} *are set to* $\omega(1,1) = 1$ *and* $\omega(1,0) = 1$, *respectively. This results in the sequence* 0 1 1 *in the domain weighted bit representation for known values of the third attribute (*a_3*); the same sequence for tuple* $[3, \varnothing, 1]$ *is obtained in* `EBU` *(Figure 5.6) and* `IBU` *(Figure 5.9) as well. Now let us look at an example of the deviation from the encoding presented before. The second attribute (*a_2*) of* $[3, \varnothing, 1]$ *has an unknown value. In this case, the corresponding bit in bitmap* B_{21} *is set to*

$$\omega(\varnothing, 1) = \frac{\|domain(a_2)\| - \|domain_{<1}(a_2)\|}{\|domain(a_2)\|} = \frac{5-1}{5} = \frac{4}{5},$$

similarly the corresponding bit in B_{22} *is set to*

$$\omega(\varnothing, 0) = \frac{\|domain(a_2)\| - \|domain_{<0}(a_2)\|}{\|domain(a_2)\|} = \frac{5-0}{5} = \frac{5}{5}.$$

This leads to the sequence $\frac{4}{5}$ $\frac{5}{5}$ *for tuple* $[3, \varnothing, 1]$ *in the domain weighted bit representation for known values of the second attribute (*a_2*).* ∘

• Domain Weighted Bitmap Construction for the NULL Attribute Values. Let $B_{i\varnothing}$ denote the domain weighted bitmap for the unknown (NULL) values of the ith attribute, $domain(i)$ be the domain of attribute i, and $domain_{\leq X}(i)$ represent the set of values in the domain of i that are less than or equal to X, where $X \neq \varnothing$. If t_i represents an unknown value of attribute i in tuple $t = (t_1, t_2, \ldots, t_d)$, then the corresponding bit in bitmap $B_{i\varnothing}$ is set using two of the $\omega(X, \varnothing)$ encodings described in Section 5.2.2: (1) if $X \neq \varnothing$, then

$$\omega(X, \varnothing) = \frac{\|domain_{\leq X}(i)\|}{\|domain(i)\|},$$

and (2) if $X = \varnothing$, then

$$\omega(X, \varnothing) = 0.5.$$

Example 21. *In Figure 5.11, the second attribute (a_2) of tuple $[3, \varnothing, 1]$ has an unknown value. In this scenario, the corresponding bit in the bitmap for unknown values of the second attribute ($B_{2\varnothing}$) will be set to $\omega(\varnothing, \varnothing) = 0.5$. On the other hand, the first attribute (a_1) of tuple $[3, \varnothing, 1]$ does not have a NULL value. In this case, the corresponding bit in bitmap $B_{1\varnothing}$ for unknown values of the first attribute is set to*

$$\omega(3, \varnothing) = \frac{\|domain_{\leq 3}(a_1)\|}{\|domain(a_1)\|} = \frac{4}{5}.$$

∘

• **Bitmap Representation of Tuples with No Unknown Values.** Like the `IBU` bitmap structure (Section 5.3.2), the `WB` index also contains a bitmap that represents the set of all *complete* tuples (with no NULL values) in the data source. As before, this is denoted by B_C.

Example 22. *Figure 5.11 illustrates this: in this example, B_C is 0011 as the first two tuples have NULL values, but the last two tuples do not have NULL values.* ∘

It is important to note that duplicate attribute values in a dataset lead to identical $\omega(X, Y)$ weights in the corresponding `WB` structure. Hence, the $\omega(X, Y)$ values can potentially be calculated only once for each distinct value in the dataset and these precomputed weights can then be reused in building the entire `WB` index structure.

Example 23. *In Figure 5.11, the $\omega(1, \varnothing)$ value corresponding to the first attribute (a_1) of tuple $[1, 1, 0]$ in the bitmap for unknown values of the first attribute ($B_{1\varnothing}$) is $\frac{2}{5}$, which is the same as the $\omega(1, \varnothing)$ value corresponding to the first attribute (a_1) of tuple $[1, 0, 0]$ in the bitmap $B_{1\varnothing}$.* ∘

5.4.2 SS Query Processing using Weighted-Bitmaps (SS-WB)

The proposed `SS-WB` algorithm executes domain weighted `SS` queries using the bitmaps in the `WB` structure by replacing the logical bitwise AND/OR operations in

SS-EBU and SS-IBU with their equivalent element-wise multiplication operations for probabilistic domains and by substituting cardinality computation with summation.

SS-WB performs the following steps on each tuple $t = (t_1, t_2, \ldots, t_d)$ in the dataset:

Case I. If tuple t has any unknown (NULL) attribute values, i.e. t is incomplete ($t_1 = \varnothing \vee t_2 = \varnothing \vee \ldots \vee t_d = \varnothing$), then SS-WB performs as follows:

1. The algorithm computes weighted bit-slice $P_w = B_{1q_1} \odot B_{2q_2} \odot \ldots \odot B_{dq_d}$, where $\odot$ represents the element-wise multiplication operation. This is the probabilistic equivalent of performing the bitwise AND operation between the bitmaps. If $t_i \neq \varnothing$, then B_{iq_i} is the domain weighted bitmap for the q_ith distinct value at attribute i, else, if $t_i = \varnothing$, then B_{iq_i} is the domain weighted bitmap, $B_{i\varnothing}$, for the unknown (NULL) values of the ith attribute.

   The weighted bit-slice P_w has the property that the nth position is set to a weight $dom_{wp}(n,t) > 0$ if and only if the nth tuple has a value in each attribute that is *likely* to be greater than or equal to the value of the corresponding attribute in t. Thus, if the nth position in P_w is set to $dom_{wp}(n,t) > 0$, the nth tuple *potentially-dominates* tuple t *with a weight* $dom_{wp}(n,t)$.

2. Next, the algorithm computes $\mathcal{R}(t) = summation(P_w) - P_w.t$, where $\mathcal{R}(t)$ is the total weight with which the tuples in the dataset *potentially-dominate* t, $summation(P)$ gives the sum of the weights in bit-slice P_w, and $P_w.t$ corresponds to the weight $dom_{wp}(t,t)$. $\mathcal{R}(t)$ does not include $P_w.t$ to avoid considering *domain-weighted-potential-dominance* of t by itself.

3. Finally, tuple t is inserted into the appropriate *skyline stratum* $S_i \in \mathcal{S}$ based on the value of $\mathcal{R}(t)$.

Case II. If, on the other hand, tuple t has no NULL values, i.e. t is complete ($t_1 \neq \varnothing \wedge t_2 \neq \varnothing \wedge \ldots \wedge t_d \neq \varnothing$), then SS-WB performs the following steps:

1. `SS-WB` computes weighted bit-slice P_w as in Case I, Step 1.

2. Next, it computes $P_C = P_w \odot B_C$, where B_C is the bitmap that represents the set of tuples with no missing values. P_C has the property that the nth position is set to 1 if and only if the nth tuple has no missing attribute values and has value in each attribute that is greater than or equal to the value of the corresponding attribute in t. If the nth bit is set to 1, then this indicates that the nth tuple is complete and may *strictly-dominate* tuple t.

3. The algorithm then calculates the number, $\mathcal{C}(t)$, of tuples that have no missing values and may *strictly-dominate* tuple t as $\mathcal{C}(t) = summation(P_C) - 1$. $\mathcal{C}(t)$ is one less than the summation of P_C to avoid considering *strict-dominance* of tuple t by itself.

4. If $\mathcal{C}(t) = 0$, there are no complete tuples in the dataset that *strictly-dominate* tuple t. Thus, `SS-WB` can

   (a) reuse already computed weighted bit-slice P_w to calculate $\mathcal{R}(t)$ as shown in Case I, Step 2 and, then,

   (b) insert tuple t into the appropriate *skyline stratum* $S_i \in \mathcal{S}$ based on $\mathcal{R}(t)$.

5. If $\mathcal{C}(t) > 0$, then there is a possibility that there is a complete tuple in the dataset which *strictly-dominates* tuple t. Thus, the algorithm confirms the existence of such a complete tuple in the dataset by carrying out the following element-wise multiplication operations:

   (a) It computes weighted bit-slice $O_w = 1 - [(1 - B_{1q_1-1}) \odot (1 - B_{2q_2-1}) \odot \ldots \odot (1 - B_{dq_d-1})]$. This represents the probabilistic equivalent of performing the bitwise OR operation between the bitmaps. The element-wise

multiplication operation is carried out on the preceding domain weighted bitmap of B_{iq_i}, which corresponds to the smallest value that is larger than t_i. If there is no preceding domain weighted bitmap for attribute i, i.e. if q_i corresponds to the first bitmap, then bitmap B_{i0} is set to 0. The operation $1 - B_{iq_i-1}$ indicates that the subtraction with 1 is performed on every element in bitmap B_{iq_i-1}.

O_w has the property that the nth position is set to a value > 0 if and only if the nth tuple in the dataset has some of its attribute values that are *likely* to be greater than the value of the corresponding attribute in t.

(b) `SS-WB` then computes $O_C = O_w \odot B_C$, where B_C denotes the bitmap that represents the set of tuples with no NULL values. O_C has the property that the nth bit is set to 1 if and only if the nth tuple has no missing values and has some of its attribute values greater than the corresponding attribute values in tuple t.

(c) Next, `SS-WB` combines P_C computed in Case II, Step 2 with O_C to obtain $D = P_C \odot O_C$.

Note that, if the nth position in D is set to 1, then this indicates that the nth tuple is complete and *strictly-dominates* tuple t.

(d) If $summation(D) = 0$, i.e. there are no complete tuples in the dataset that *strictly-dominate* t, then `SS-WB`

i. reuses already computed weighted bit-slice P_w to calculate $\mathcal{R}(t)$ as shown in Case I, Step 2, then,

ii. inserts tuple t into the appropriate *skyline stratum* $S_i \in \mathcal{S}$ based on the value of $\mathcal{R}(t)$.

(e) If $summation(D) > 0$, then there is at least 1 complete tuple that *strictly-*

dominates t. Thus, tuple t goes to the lowest strata that is pruned away.

A detailed pseudocode of `SS-WB` is presented in Figure 5.12.

Example 24. *Let us consider the input in Figure 5.11 and determine how* $[\varnothing, 1, 2]$ *is inserted into* $\mathcal{S}$. *Tuple* $[\varnothing, 1, 2]$ *contains a NULL value in the first attribute* (a_1). *Hence, we compute* P_w *by performing the element-wise multiplication operation between the bitmap of the unknown value in the first attribute and the bitmaps for the known values (1, 2) in the second* (a_2) *and third* (a_2) *attributes:*

$$\begin{aligned} P_w &= B_{1\varnothing} \odot B_{21} \odot B_{31} \\ &= 0.5\ \frac{4}{5}\ \frac{2}{5}\ \frac{2}{5} \odot 1\ \frac{4}{5}\ 1\ 0 \odot 1\ 0\ 0\ 0 \\ &= 0.5\ 0\ 0\ 0. \end{aligned}$$

Next, we compute $\mathcal{R}([\varnothing, 1, 2])$*:*

$$\mathcal{R}([\varnothing, 1, 2]) = summation(P_w) - P_w.[\varnothing, 1, 2] = 0.5 - 0.5 = 0.$$

Consequently, as shown in Figure 5.13, $[\varnothing, 1, 2]$ *is inserted into* skyline stratum S_0.

Let us now consider a scenario in which a tuple is complete. It can be observed that tuple $[1, 0, 0]$ *has no missing values. Hence, we first compute* P_w *by performing the element-wise multiplication operation between the bitmaps* B_{12}, B_{22} *and* B_{33}*:*

$$P_w = \frac{4}{5}\ 1\ 1\ 1 \odot 1\ \frac{5}{5}\ 1\ 1 \odot 1\ 1\ 1\ 1 = \frac{4}{5}\ 1\ 1\ 1.$$

Then, we compute P_C*:*

$$P_C = P_w \odot B_C = \frac{4}{5}\ 1\ 1\ 1 \odot 0\ 0\ 1\ 1 = 0\ 0\ 1\ 1.$$

Next, we compute $\mathcal{C}([1, 0, 0])$*:*

$$\mathcal{C}([1, 0, 0]) = summation(P_C) - 1 = 1$$

Algorithm 3: SS-WB($\mathcal{D}, A_S$, WB)

Input:

$\mathcal{D}$: An incomplete dataset, A_S: A set of skyline attributes

WB: Weighted-Bitmaps index structure

Output:

$\mathcal{S}$: A set of *skyline strata*

Procedure:

for each tuple $t = (t_1, t_2, \ldots, t_d)$ in $\mathcal{D}$ **do**

if $t_1 = \varnothing \vee t_2 = \varnothing \vee \ldots \vee t_d = \varnothing$ **then**

$P_w := \texttt{WB}.B_{1q_1} \odot \texttt{WB}.B_{2q_2} \odot \ldots \odot \texttt{WB}.B_{dq_d}$ (use $\texttt{WB}.B_{i\varnothing}$ if $t_i = \varnothing$)

$\mathcal{R}(t) := summation(P_w) - P_w.t$

Insert tuple t into the appropriate *skyline stratum* S_i in $\mathcal{S}$ based on $\mathcal{R}(t)$

end if

if $t_1 \neq \varnothing \wedge t_2 \neq \varnothing \wedge \ldots \wedge t_d \neq \varnothing$ **then**

$P_w := \texttt{WB}.B_{1q_1} \odot \texttt{WB}.B_{2q_2} \odot \ldots \odot \texttt{WB}.B_{dq_d}$ (use $\texttt{WB}.B_{i\varnothing}$ if $t_i = \varnothing$)

$P_C := P_w \odot \texttt{WB}.B_C$

$\mathcal{C}(t) := summation(P_C) - 1$

if $\mathcal{C}(t) = 0$ **then**

$\mathcal{R}(t) := summation(P_w) - P_w.t$

Insert t into the appropriate *skyline stratum* S_i in $\mathcal{S}$ based on $\mathcal{R}(t)$

else

$O_w = 1 - [(1 - \texttt{WB}.B_{1q_1-1}) \odot (1 - \texttt{WB}.B_{2q_2-1}) \odot \ldots \odot (1 - \texttt{WB}.B_{dq_d-1})]$

$O_C := O_w \odot \texttt{WB}.B_C$

$D := P_C \odot O_C$

if $summation(D) = 0$ **then**

$\mathcal{R}(t) := summation(P_w) - P_w.t$

Insert t into the appropriate *skyline stratum* S_i in $\mathcal{S}$ based on $\mathcal{R}(t)$

else

Tuple t is *strictly-dominated* and it is not inserted into $\mathcal{S}$

end if

end if

end if

end for

return $\mathcal{S}$

Figure 5.12: The SS-WB algorithm

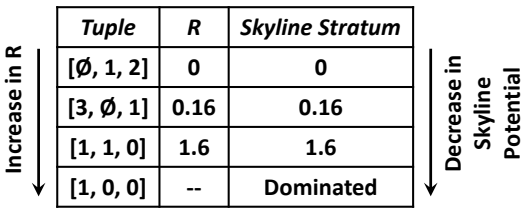


Figure 5.13: An example result of an SS query based on domain weighted potential dominance (dom_{wp}) obtained via the SS-WB algorithm

Since $\mathcal{C}([1,0,0]) > 0$*, we compute* O_w *by using the bitmaps* B_{11}*,* B_{21} *and* B_{32}*:*

$$\begin{aligned} O_w &= 1 - [(1 - B_{11}) \odot (1 - B_{21}) \odot (1 - B_{32})] \\ &= 1 - [\frac{3}{5}\ 0\ 1\ 1 \odot 0\ \frac{1}{5}\ 0\ 1 \odot 0\ 0\ 1\ 1] \\ &= 1 - [0\ 0\ 0\ 1] = 1\ 1\ 1\ 0. \end{aligned}$$

Then, we compute O_C*:*

$$O_C = O_w \odot B_C = 1\ 1\ 1\ 0 \odot 0\ 0\ 1\ 1 = 0\ 0\ 1\ 0.$$

Next, we compute D *using* P_C *and* O_C*:*

$$D = P_C \odot O_C = 0\ 0\ 1\ 1 \odot 0\ 0\ 1\ 0 = 0\ 0\ 1\ 0.$$

Since $summation(D) > 0$*, tuple* $[1,0,0]$ *is* strictly-dominated*. Hence, as shown in Figure 5.13,* $[1,0,0]$ *goes to the lowest strata that is pruned away.*

Figure 5.13 also shows the complete set of skyline strata *returned by the* SS-WB *algorithm for the given input dataset.* ∘

5.5 Worst-Case Time Complexity

Let us consider a dataset $\mathcal{D}$ with d skyline attributes and n tuples. For both SS-EBU and SS-IBU, the number of logical bitwise operations for a given tuple is highest when the tuple does not contain a NULL value. In this case, both SS-EBU and SS-IBU first check whether a tuple t is strictly-dominated by some other tuple in $\mathcal{D}$,

and if t is not strictly-dominated by any other tuple in $\mathcal{D}$, the unweighted SS algorithms map tuple t into an appropriate stratum. Therefore, the worst-case complexity of both SS-EBU and SS-IBU is $O(n \times [d \times (T_{OR}(n) + T_{AND}(n)) + T_{lookup}(l)])$, where $T_{OR}(n)$ and $T_{AND}(n)$ are the time taken for the bitwise OR and AND operations involving n-bit bitmaps, respectively, and $T_{lookup}(l)$ is the time taken to insert one tuple into the appropriate stratum among the *skyline strata* containing $l \leq n$ layers.

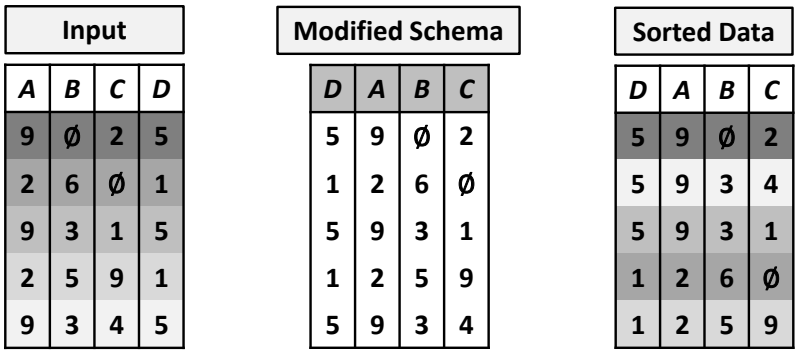
The domain weighted SS-WB algorithm replaces the logical bitwise operations with their equivalent element-wise multiplication operations. For the SS-WB algorithm, the number of element-wise multiplication operations for a given tuple is highest when the tuple does not contain a NULL value. In this case, SS-WB first checks whether a tuple t is strictly-dominated by some other tuple in $\mathcal{D}$, and if t is not strictly-dominated by any other tuple in $\mathcal{D}$, it maps tuple t into an appropriate skyline stratum. Hence, the worst-case complexity of SS-WB is $O(n \times [d \times T_{\odot}(n) + T_{lookup}(l)])$, where $T_{\odot}(n)$ is the time taken for the element-wise multiplication operations involving weighted bitmaps of size n and, like before, $T_{lookup}(l)$ is the time taken to insert one tuple into the appropriate stratum among the *skyline strata* containing $l \leq n$ layers.

5.6 Optimizations

In this section, two optimization techniques are introduced.

5.6.1 Data Sorting and Sub-Result Reuse

Let us consider a dataset with schema R(A, B, C, D) and that contains tuples $t_1[9, \varnothing, 2, 5]$, $t_2[2, 6, \varnothing, 1]$, $t_3[9, 3, 1, 5]$, $t_4[2, 5, 9, 1]$ and $t_5[9, 3, 4, 5]$ (Figure 5.14(a)). Here, tuples t_3 and t_5 overlap in three attributes, tuple t_1 overlaps with tuples t_3 and t_5 in two attributes, and tuples t_2 and t_4 overlap in two attributes. It is easy to see that, for this dataset, the same sequence of bitwise operations may need to



(a) (b) (c)

Figure 5.14: (a) Input dataset; (b) The schema of the dataset is first modified by sorting the attributes from the **most** overlapping attribute to the **least** overlapping attribute; (c) The tuples are then sorted in the descending order of the attribute values: unknown values (∅) are treated as larger than the largest value in the domains of the attributes

be performed for multiple tuples. We note that it may be possible to significantly reduce the amount of redundant work when tuples have overlaps in their attribute values. This is possible if we could reuse the intermediate bitwise operations between the bitmaps of attribute values of one tuple for processing other tuples in the dataset.

However, a naive way of implementing this through indexing and caching of all intermediary bitwise operation results would require an inordinate amount of space and likely provide poor cache utilization. Instead, we propose to group similar tuples together by (a) first, sorting the attributes of the data from the most overlapping to the least and (b) then, sorting the tuples in the dataset in the descending order of its attribute values, with unknown known values (represented by ∅) being treated as larger than the largest value in the domains of the attributes.

Example 25. *Figure 5.14 shows an example. The schema in Figure 5.14(a) would be modified to* $\texttt{R}(\texttt{D}, \texttt{A}, \texttt{B}, \texttt{C})$ *(Figure 5.14(b)) and the tuples would be sorted as* $[5, 9, \varnothing, 2]$, $[5, 9, 3, 4]$, $[5, 9, 3, 1]$, $[1, 2, 6, \varnothing]$ *and* $[1, 2, 5, 9]$ *(Figure 5.14(c)).*

This order helps the `SS-EBU` *and* `SS-IBU` *algorithms leverage sub-results of bitwise*

operations between the bitmaps of successive attribute values of one tuple for processing tuples that immediately follow it[2]*. For instance, the result of the bitwise operations between the bitmaps of the value sequence* $(5, 9)$ *in Figure 5.14(c), obtained as a sub-result of the bitwise operations between the bitmaps of tuple* $[5, 9, \varnothing, 2]$*, can be reused when processing the value sequence* $(5, 9, 3)$ *in tuple* $[5, 9, 3, 4]$*. Similarly, the result of the bitwise operations between the bitmaps of the value sequence* $(5, 9, 3)$*, cached as a sub-result when processing tuple* $[5, 9, 3, 4]$*, would be reused while processing the subsequent tuple* $[5, 9, 3, 1]$*. Note that tuple* $[1, 2, 6, \varnothing]$ *shown in Figure 5.14(c) has no overlap with the previous tuple* $[5, 9, 3, 1]$*. At this point, the previous cached sub-results can be thrown away. This ensures that the amount of sub-results that need to be cached is minimal.*

We evaluate the impact of sub-result reuse in Section 5.8.5.

5.6.2 Compressed Domain Execution

For efficient processing (i.e., to reduce I/O and CPU costs), the bitmaps in the `EBU` and `IBU` bitmap structures are compressed and stored using the Enhanced Word-Aligned Hybrid (EWAH) compression technique [40]. Moreover, all the logical operations are carried out over these compressed bitmaps to avoid having to uncompress bitmaps and to accelerate bitwise logical operations. More specifically, in our implementation, we constructed the unweighted bitmap index structures (`EBU` and `IBU`) and executed compressed domain operations using the JavaEWAH library, which is an EWAH compressed variant of the Java bitset class. The JavaEWAH library is available for download at `https://code.google.com/p/javaewah/`.

On the other hand, the domain weighted `WB` structure that contains weights in ad-

[2]As we also see in Section 5.8.5, such reordering leads to better compression of the columns – thus this order should be the preferred data ordering even in absence of Strata-Skyline queries.

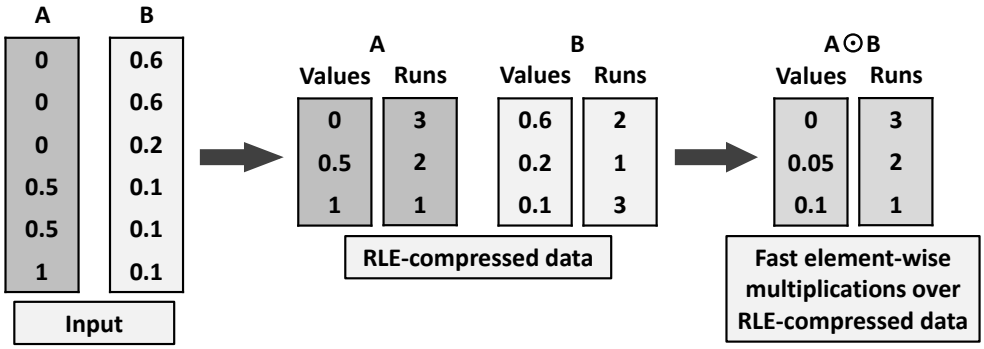


Figure 5.15: The number of element-wise multiplication operations can be largely reduced when these operations are executed over RLE-compressed data

dition to 0s and 1s is constructed and stored using the Run-Length-Encoding (RLE) compression technique. The implementation for the RLE compression based `WB` structure was adapted from `http://people.brunel.ac.uk/~csstnns/docs/RLE.java`. Also, in order to leverage compressed domain operations, we implemented our own method to carry out fast element-wise multiplication operations over RLE-compressed `WB` index structures. The number of element-wise operations can be largely reduced when these operations are executed over RLE-compressed structures.

Example 26. *Let us consider the example domain weighted input in Figure 5.15. This input is compressed using the Run-Length-Encoding (RLE) compression technique. As shown in the figure, the RLE-compressed data stores all distinct values in the inputs, A and B, along with the lengths of their corresponding runs. For example, the value* 0 *in input A repeats three times, hence it results in a run of length* 3 *for value* 0 *in the RLE-compressed version of A. As can be observed, the number of element-wise multiplication operations between A and B can be largely reduced when these operations are executed over the RLE-compressed versions of A and B. For instance, when computing* $A \odot B$ *(where* $\odot$ *is the element-wise multiplication operator), the number of the multiplications with 0s in column A can be reduced from three to one when this operation is carried out on the RLE-compressed version of the data.*

Hence, our method can help accelerate element-wise operations. It also largely avoids uncompressing the bitmaps in the WB structure during Strata-Skyline query processing. Moreover, if the attribute values in a dataset are sorted using the method described in the previous sub-section and then compressed using the techniques described above, it leads to a clustering of weights and longer runs of these values in the corresponding bitmap index structures. As a result, such reordering leads to a better compression of the bitmaps and hence, as shown in Sections 5.8.5 and 5.8.6, helps achieve faster processing of Strata-Skyline queries.

5.7 Possible Extensions

This section outlines some future research directions for Strata-Skyline queries.

5.7.1 Strata-Skyline Queries with Data Updates

When a new tuple comes into the database or a tuple is removed, in addition to finding the stratum of the new tuple and seeing if the strata of any of the existing tuples is impacted, we also need to update the bitmap index structures. While the details of the update process are left for future work, it is important to note that updates on the bitmaps are highly localized and therefore can be performed very efficiently with time complexity independent of the data size.

5.7.2 Top-k Strata-Skyline Queries

Another interesting topic related to SS queries is the efficient computation of only the top few skyline strata with the highest skyline potentials. One possible naive way of addressing this would be to use the proposed SS algorithms to stratify the entire input dataset and then output only the top-k skyline strata. This method of answering top-k SS queries can prove to be wasteful, especially if the required number of skyline

strata (i.e. k) is very small. Hence, the key question here is whether it would be possible to find the top-k skyline strata without stratifying all the tuples in a dataset. The study of this extension is left for future work.

5.8 Performance Evaluation

This section experimentally evaluates the performance of the proposed Strata-Skyline (SS) algorithms. We compare the SS-EBU, SS-IBU and SS-WB algorithms to an alternative method, SS-BNL, that stratifies the tuples into their respective skyline layers by performing exhaustive tuple-to-tuple comparisons.

• **Synthetic Datasets.** Evaluations were carried out on 47 different synthetic datasets and real datasets. Synthetic datasets were generated based on correlated, anti-correlated and independent distributions[3], as described in [13]. We remove independently randomly selected attribute values to induce a certain percentage of data incompleteness (I) in order to test the performance of our algorithms. Removed values represent missing/unknown data in the datasets. The cardinality of the datasets (n) was varied between $10,000$ and $1,000,000$ tuples. The data incompleteness (I) considered was 1% (i.e., 1 in 100 attribute values in a data source is missing), 10%, 20%, 50% and 65%. The dimensionality (d) of the skyline attribute set was varied between 2 and 5.

• **MovieLens Dataset.** MovieLens[4] dataset contains about 10 million user ratings related to $\sim 10,000$ movies belonging to 18 different genres rated by about $72,000$ users. The MovieLens dataset is naturally incomplete because not all users have rated all the movies and genres, hence many user ratings are missing. From this, we generated datasets of various data incompleteness (I) by selecting different subsets of

[3]Dataset generator available at http://randdataset.projects.postgresql.org/.

[4]Downloaded from http://grouplens.org/datasets/movielens/.

genres rated by 69,878 users. For instance, a dataset of low incompleteness ($I = 1\%$) was obtained by considering popular genres such as *Sci-Fi, Crime, Adventure, and Romance*. On the other hand, a dataset of high incompleteness ($I = 38\%$) was obtained by combining less popular genres like *Documentary, Film-Noir, Western, and Animation*. The percentages of incompleteness (I) considered were 1%, 12%, 26%, and 38%.

• **DBLP Dataset.** From the DBLP[5] dataset, we parsed publication statistics of about 265,329 authors. The information collected for each author has 4 attributes representing statistics related to the total number of papers published, number of co-author relationships, number of citations, and the number of publication venues. This information was recorded in a table of the form: *Authors (authorID, papers, venues, citations, coauthors)*. We removed randomly selected values in order to test the performance of our algorithms. The removed attribute values represent missing statistics about the authors.

• **Evaluation Setup.** All experiments were conducted on a machine running Windows 7 operating system, with an Intel Core i5 3.10GHz processor and 8GB RAM. The `SS-EBU`, `SS-IBU`, `SS-WB` and `SS-BNL` algorithms are all implemented in Java. The datasets are stored on disk using the MySQL database[6]. The `EBU` and `IBU` bitmap structures are constructed using JavaEWAH 0.8.4[7], a word-aligned compressed variant of the Java bitset class, which supports compressed domain bitwise operations. Whereas, the `WB` bitmap structure is built using Run-Length-Encoding (RLE) based compression[8] to support element-wise multiplications over RLE-compressed data.

`SS-EBU`, `SS-IBU` and `SS-WB` execute `SS` queries by reading tuples once from the

[5]Downloaded from `http://arnetminer.org/citation`

[6]`http://www.mysql.com/`

[7]Available at `https://code.google.com/p/javaewah/`

[8]Implementation adapted from `http://people.brunel.ac.uk/~csstnns/docs/RLE.java`

disk through the MySQL database. This is the only time the algorithms access the disk, the rest of the structures are stored in-memory. As we see later, the cost of reading the data from the disk (against the cost of dominance checks) is negligible. For example, it takes only $\sim$ 291 ms to read the DBLP dataset (n= $265,329$ tuples, d=4) from the database. As we also see later, 1GB was more than enough to store the data and the `EBU`, `IBU` and `WB` bitmap structures even for the largest considered dataset. Therefore, we made available to the Java machine only 1GB for the `SS-EBU`, `SS-IBU` and `SS-WB` algorithms. The time taken to read the bitmap structures from the disk into memory is negligible compared to the `SS` query time. For the aforementioned DBLP dataset, for instance, the time taken to load the `EBU` structure from the disk is $\sim$ 602 ms, whereas the load time for the `IBU` structure is $\sim 1,344$ ms.

We compared `SS-EBU`, `SS-IBU` and `SS-WB` against an exhaustive algorithm, `SS-BNL`, which uses a block-nested-loops strategy to enumerate all tuple-pairs to perform the necessary potential dominance checks, but avoids generating all tuple-pairs for a tuple that is found to be strictly-dominated. In other words, `SS-BNL` avoids wasteful strict dominance checks. When the dataset has no incomplete tuples, `SS-BNL` behaves like the BNL skyline operator [13]. 1GB of memory was also made available to the `SS-BNL` algorithm. Since this is sufficient to hold the entire dataset in memory, in its first pass, the algorithm loads the data into memory as a single block and then, in its second pass over the data, it produces tuple-pairs for dominance checks.

Each experiment is run five times and the execution times reported are the averages of the five runs. Unless otherwise specified, sub-result reuse is turned off and the data is not ordered in any particular way.

• **Performance Metrics.** As is common in assessing skyline algorithms, we used query execution time as the major metric in evaluating the our techniques. Query execution time is the duration from the time an algorithm starts to the time it returns

I	1%	12%	26%	38%
SS-EBU (s)	0.50	0.50	0.50	0.50
SS-IBU (s)	0.51	0.51	0.51	0.52

(a) Bitmap structure construction time

I	1%	12%	26%	38%
SS-EBU (s)	7.37	5.86	4.89	4.04
SS-IBU (s)	9.55	6.34	4.09	3.08
SS-BNL (s)	28.70	225.84	425.60	480.23

(b) Query execution time

Figure 5.16: Effect of incompleteness, I (MovieLens data, $n = 69K$, $d = 4$)

the skyline strata of an entire dataset. Furthermore, we report other evaluation metrics such as the index construction time and the amount of main memory utilized by the EBU and IBU index structures.

5.8.1 Effect of Incompleteness

In this subsection, we study the effect of data incompleteness (I) on the SS-EBU, SS-IBU and SS-BNL algorithms.

Evaluation on the MovieLens and DBLP Datasets

Figures 5.16 and 5.17 show the performance of the various alternatives as the percentage of incompleteness (I) is varied for the MovieLens and DBLP data sets. Since the results are similar, we only discuss the DBLP results in Figure 5.17.

As can be seen, the time spent by SS-EBU and SS-IBU on bitmap construction (Figure 5.17(a)) is negligible compared to the gains they help achieve during query processing (Figure 5.17(b)). Also, the bitmap structure needs to be built only once and can be reused for different Strata-Skyline (SS) queries on the dataset. Therefore, the overhead in constructing the bitmap index structures can be further amortized over time.

Figure 5.17(a) indicates that the change in data incompleteness has very little effect on the time taken to construct the bitmap structures. This is because the

amount of work that goes into building the bitmaps is primarily dependent on the cardinality of the datasets. This aspect is further analysed in Section 5.8.2. Also, in general, the time taken by SS-IBU to build the Implicit-Bitmaps-for-Unknowns (IBU) index structure is slightly higher when compared to the time taken by SS-EBU to build the Explicit-Bitmaps-for-Unknowns (EBU) index structure. This occurs because IBU incurs an added overhead to *implicitly* indicate missing values and mark the set of tuples with no missing values (Section 5.3.2).

Figure 5.17(b) shows the effect of data incompleteness on query execution time. Overall, the proposed Strata-Skyline (SS) algorithms perform multiple orders of time faster than the SS-BNL algorithm. The main reason for this behaviour is that the SS algorithms carry out the stratification process using fast bitwise operations over compressed bitmap structures as opposed to expensive tuple-to-tuple comparisons. As can be observed, the execution time of the two SS algorithms reduces as the percentage of incompleteness increases, whereas the SS-BNL algorithm behaves in an inverse manner. The main reason for this is that, with more incomplete data, the number of bitmaps accessed by the SS algorithms reduces because the bitwise operations to stratify a tuple is carried out by using bitmaps only related to the know attribute values of the tuple being stratified.

The execution time of SS-BNL is much higher due to the fact that it stratifies tuples into their respective skyline layers by performing a large number of tuple-to-tuple comparisons. For example, in the case where SS-BNL takes $6,255.71$ s to stratify the data ($I = 20\%$ in Figure 5.17(b)), $\sim 3,174.77$ s is spent on dominance checking and $\sim 3,078.72$ s is spent on repeatedly scanning the dataset in-memory. Note, when data incompleteness is low, SS-BNL works similar to block-nested-loop skyline [13] and thus, is able to avoid a large portion of the comparisons. When data incompleteness increases, there are lesser number of complete tuples; this reduces

I	1%	10%	20%	50%	65%
SS-EBU (s)	1.42	1.51	1.45	1.47	1.42
SS-IBU (s)	1.56	1.51	1.50	1.46	1.45

(a) Bitmap structure construction time

I	1%	10%	20%	50%	65%
SS-EBU (s)	111.34	86.49	74.98	42.10	27.26
SS-IBU (s)	124.71	87.53	65.89	19.40	7.62
SS-BNL (s)	427.57	3,729.36	6,255.71	9,346.89	8,530.27

(b) Query execution time

Figure 5.17: Effect of data incompleteness, I (DBLP dataset, $n = 265K$, $d = 4$)

pruning opportunities and SS-BNL works even slower.

Another observation that can be made from Figure 5.17(b) is that SS-IBU runs faster than SS-EBU as the percentage of incompleteness increases. This is mainly due to the fact that, with more incomplete data, the SS-IBU algorithm has to perform lesser number of bitwise operations to find the set of tuples that can potentially-dominate a particular tuple. SS-IBU utilizes the IBU index structure that *implicitly* indicates missing values, thus preventing it from incurring the overhead of performing the additional bitwise OR operations that the SS-EBU algorithm performs during the stratification process. On the other hand, SS-EBU cannot avoid these bitwise OR operations, thus causing it to relatively slow down as the number of missing values in the dataset increases.

At very low incompleteness ($I = 1\%, 10\%$ in Figure 5.17(b)), SS-EBU has a better execution time than the SS-IBU algorithm. This is due to the fact that SS-EBU has to perform lesser number of bitwise operations to find the strict dominance relations between complete tuples in a dataset. SS-IBU, on the other hand, has to perform additional bitwise AND operations using the bitmap that denotes the set of tuples

I	1%	10%	20%	50%	65%
SS-EBU (ms)	723.6	695.8	692.8	686.2	664.4
SS-IBU (ms)	748.8	730.4	720.6	726.8	714.4

(a) Bitmap structure construction time

I	1%	10%	20%	50%	65%
SS-EBU (s)	9.16	8.97	8.67	6.04	3.88
SS-IBU (s)	18.73	12.39	9.76	3.97	1.95
SS-BNL (s)	58.97	512.60	843.03	1,281.77	1,176.75

(b) Query execution time

Figure 5.18: Effect of incompleteness, I (Correlated data, $n = 100K$, $d = 4$)

I	1%	10%	20%	50%	65%
SS-EBU (ms)	705.2	692.4	689.4	683.0	661.6
SS-IBU (ms)	752.2	717.8	698.8	683.0	683.4

(a) Bitmap structure construction time

I	1%	10%	20%	50%	65%
SS-EBU (s)	9.43	8.83	8.73	5.61	3.78
SS-IBU (s)	18.01	12.22	9.63	3.26	1.53
SS-BNL (s)	62.23	547.44	961.22	1,357.51	1,230.33

(b) Query execution time

Figure 5.19: Effect of incompleteness, I (Independent data, $n = 100K$, $d = 4$)

with no missing values in order to find the strict dominance relations. This overhead is higher when the number of complete tuples in a dataset is more, thus preventing SS-IBU from performing as well as SS-EBU when the data incompleteness is very low.

Evaluation on Synthetic Datasets

This section studies the effect of data incompleteness on synthetic datasets with correlated, independent and anti-correlated distributions. As the percentage of in-

I	1%	10%	20%	50%	65%
SS-EBU (ms)	761.6	705.2	699.2	695.6	686.4
SS-IBU (ms)	761.2	736.6	726.8	695.8	686.2

(a) Bitmap structure construction time

I	1%	10%	20%	50%	65%
SS-EBU (s)	11.43	10.05	8.65	5.62	3.80
SS-IBU (s)	15.72	12.42	8.35	2.80	1.49
SS-BNL (s)	437.01	804.62	1,099.99	1,345.31	1,228.04

(b) Query execution time

Figure 5.20: Effect of incompleteness, I (Anti-correlated, $n = 100K$, $d = 4$)

n	$10K$	$100K$	$1000K$
SS-EBU (ms)	404.4	558.2	2,623.4
SS-IBU (ms)	417.0	562.8	2,683.4

(a) Bitmap structure construction time

n	$10K$	$100K$	$1000K$
SS-EBU (s)	0.22	3.11	251.82
SS-IBU (s)	0.23	5.28	459.82
SS-BNL (s)	4.89	443.20	46,802.31

(b) Query execution time

Figure 5.21: Effect of data cardinality, n (Independent data, $d = 2$, $I = 20\%$)

completeness (I) is varied from low to high, the performance of the SS algorithms on correlated (Figure 5.18), independent (Figure 5.19) and anti-correlated (Figure 5.20) data distributions is very similar to how they performed on the DBLP and MovieLens datasets: the time spent by SS-EBU and SS-IBU on bitmap construction (Figures 5.18(a), 5.19(a), 5.20(a)) is negligible when compared to the gains they help achieve during query processing (Figure 5.18(b), 5.19(b), 5.20(b)). Also, the SS algorithms perform multiple orders of time faster than SS-BNL on all three distributions.

Since the performances of the proposed algorithms are very similar across correlated, independent and anti-correlated data distributions, in the following sections, we show results only for datasets with independent distribution.

5.8.2 *Effect of Data Cardinality*

In this subsection, we study the performance of the SS algorithms against increases in the data cardinality (n) of datasets. The cardinalities considered were $10,000$ ($10K$), $100,000$ ($100K$) and 1 million ($1000K$) tuples per dataset.

Figure 5.21(a) shows that the index construction time scales well and goes up only by a small amount when the data cardinality increases. Also, similar to what we saw earlier in section 5.8.1, the time taken by SS-IBU to build the Implicit-Bitmaps-for-Unknowns (IBU) index structure is slightly higher when compared to the time taken by SS-EBU to build the Explicit-Bitmaps-for-Unknowns (EBU) index structure. This is because IBU incurs an added overhead to *implicitly* indicate missing values and to construct a bitmap that denotes the set of tuples with no missing values.

Figure 5.21(b) examines the performance of the algorithms in terms of how fast the tuples are processed by the various alternatives. From this result, we can see that both SS-EBU and SS-IBU can stratify tuples much faster than the SS-BNL algorithm even as the cardinality increases from $n = 10K$ to $n = 100K$ to $n = 1000K$. This is due to the fact that the SS algorithms carry out the stratification process using fast bitwise operations over compressed bitmap structures as opposed to expensive tuple-to-tuple comparisons.

5.8.3 *Effect of Data Dimensionality*

Figure 5.22 shows that SS-EBU and SS-IBU scale particularly well to SS operations on high dimensional incomplete data sources. As can be observed, the bitmap construction time (Figure 5.22(a)) is hardly affected by the change in the number of skyline attributes (d). On the other hand, the execution time gains (Figure 5.22(b)) of both SS algorithms increase against SS-BNL as the number of skyline attributes

d	2	3	4	5
SS-EBU (ms)	404.4	417.6	411.8	434.6
SS-IBU (ms)	417.0	428.0	418.0	435.2

(a) Bitmap structure construction time

d	2	3	4	5
SS-EBU (ms)	218.8	260.2	334.0	371.2
SS-IBU (ms)	233.4	267.8	304.8	328.6
SS-BNL (ms)	4,885.2	7,063.4	9,821.4	13,247.4

(b) Query execution time

Figure 5.22: Effect of dimensionality, d (Independent data, $n = 10K$, $I = 20\%$)

increases. This illustrates that the proposed algorithms clearly outperform SS-BNL in all cases and are more scalable as compared to SS-BNL. It also shows that the method of stratification using fast bitwise operations over compressed bitmap structures is effective even on datasets with high dimensions.

Figure 5.22(b) also illustrates that data dimensionality (d) has a direct impact on the relative performances of the proposed SS algorithms. In particular, the SS-EBU algorithm performs relatively better than SS-IBU when the data dimensionality is low ($d = 2, 3$), whereas the two SS algorithms exchange places when the dimensionality is high ($d = 4, 5$).

5.8.4 Sizes of the Bitmap Index Structures

This subsection examines the sizes of the bitmap index structures leveraged by the proposed algorithms. As shown by the experimental results over various settings on real and synthetic datasets (Figure 5.23), the sizes of the bitmap structures grow linearly with the cardinality (n) of the datasets. On large data (for instance, $n = 100K$), the index size increases linearly with the number of attributes (d). Also,

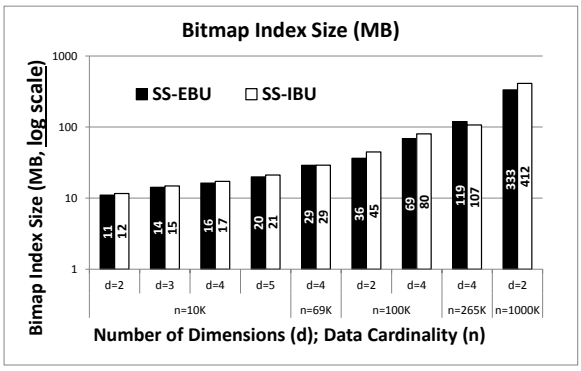


Figure 5.23: Bitmap index sizes in experiments on real and synthetic datasets

Reuse	No	Yes
`SS-EBU` (ms)	301.4	357.8
`SS-IBU` (ms)	250.8	272.0

(a) $n = 10K$ tuples

Reuse	No	Yes
`SS-EBU` (s)	5.03	2.96
`SS-IBU` (s)	2.89	2.22

(b) $n = 100K$ tuples

Figure 5.24: Effect of sub-result reuse ($d = 5$, $I = 20\%$)

except for very small datasets ($n \leq 10K$), the per tuple index requirement is only 300-800 bytes, enabling fast in-memory maintenance of bitmaps even for large datasets.

5.8.5 Effect of Sub-Result Reuse and Sorting

We investigate the effect of sub-result reuse in Figures 5.24 and 5.25. Here, the order of the data in the datasets are assumed to conform to the order discussed in Section 5.6.1.

To see the impacts of the amount of overlap and the data size on reuse, we first considered synthetic datasets, where we fixed the attribute value domains to $[1, 1000]$ and varied the number of tuples from $n = 10K$ to $n = 100K$: in the former case, each attribute value repeats ≈ 10 times, whereas in the latter, each attribute value repeats itself ≈ 100 times. As expected, for the dataset with small number of tuples and a low overlap rate, the overhead in maintaining a sub-result cache outweighs the gains obtained through the reuse of cached results (Figure 5.24(a)). On

Reuse	No	Yes
SS-EBU (s)	24.62	10.14
SS-IBU (s)	18.57	13.19

Figure 5.25: Sub-result reuse on DBLP data ($n = 265K$, $d = 4$, $I = 20\%$)

the other hand, for large datasets with high overlap rates, sub-result reuse provides significant savings, especially for SS-EBU, which has a higher potential for redundant work (Figure 5.24(b)).

These results are reconfirmed in Figure 5.25, which shows the effect of utilizing sub-result reuse on the DBLP dataset. Again, sub-result reuse provides significant savings in query execution time since the attributes contain overlaps.

Note that, a second important observation from these figures is that the SS algorithms benefit significantly from sorting of the data. Let us compare, for example, Figure 5.25 and Figure 5.17(b, $I = 20\%$): on unsorted data SS-EBU and SS-IBU run in ~ 75 s and ~ 66 s, respectively; on sorted data under the same conditions, however, they only cost ~ 25 s and ~ 19 s even if reuse is not leveraged. As discussed in Section 5.6.1, this is because the reordering of the data leads to better compression and thus, speeds up the compressed bitwise operations.

5.8.6 Effect of Domain Weighted Potential Dominance

In this subsection, the effect of using domain weighted potential dominance is investigated by examining the performance of the SS-WB algorithm on the MovieLens dataset. SS-WB is evaluated on both discrete and continuous domain distributions.

Evaluations based on the Discrete Domain Distribution Assumption

In this subsection, the $\omega(X, Y)$ weight values for the SS-WB algorithm is evaluated based on the assumption that the distribution of the values in the domains of the

I	1%	12%	26%	38%
SS-WB [No RLE] (ms)	634.0	632.0	663.0	663.5
SS-WB (ms)	640.0	686.5	710.0	686.5

(a) Domain weighted bitmap structure construction time

I	1%	12%	26%	38%
Obtained using domain weighted potential dominance				
SS-WB [No RLE] (s)	130.07	98.90	64.18	51.96
SS-WB (s)	554.04	642.69	474.67	407.88
SS-BNL (s)	14.63	143.59	291.13	330.46
Obtained using unweighted potential dominance				
SS-EBU (s)	7.37	5.86	4.89	4.04
SS-IBU (s)	9.55	6.34	4.09	3.08

(b) Query execution time

Figure 5.26: Effect of using domain weighted potential dominance on data that is not sorted (MovieLens data, $n = 69K$, $d = 4$)

skyline attributes is discrete and uniform (Section 5.2.3).

Figure 5.26 shows results on the MovieLens dataset that is not sorted according to any order, whereas Figure 5.27 shows results in which the data is sorted according to the order discussed in Section 5.6.1. SS-WB was studied under two implementations: a) SS-WB [No RLE]: this implementation of SS-WB utilizes WB index structures that are not compressed using Run-Length-Encoding (RLE), hence, the element-wise multiplication operations required to compute the skyline strata are carried out on uncompressed domain, and b) SS-WB: this implementation leverages RLE-compressed WB structures, therefore, it uses compressed element-wise multiplication operations.

The performance of SS-WB is compared against a domain weighted version of the SS-BNL algorithm. In this case, SS-BNL uses the domain weighted potential dominance definition and, like before, a block-nested-loops strategy to enumerate

I	1%	12%	26%	38%
SS-WB [No RLE] (ms)	679.8	652.2	658.4	652.2
SS-WB (ms)	689.6	664.4	667.8	670.8

(a) Domain weighted bitmap structure construction time

I	1%	12%	26%	38%
Obtained using domain weighted potential dominance				
SS-WB [No RLE] (s)	178.02	127.92	77.40	59.69
SS-WB (s)	2.62	3.22	2.30	2.18
SS-BNL (s)	14.23	120.53	231.37	263.58
SS-BNL [Naive] (s)	451.99	401.83	357.29	349.30
Obtained using unweighted potential dominance				
SS-EBU (s)	1.12	1.53	1.57	1.46
SS-IBU (s)	1.40	1.54	1.03	0.87

(b) Query execution time

Figure 5.27: Effect of using domain weighted potential dominance on data that is sorted (MovieLens data, $n = 69K$, $d = 4$)

all tuple-pairs to perform the necessary potential dominance checks. As explained earlier, SS-BNL avoids generating all tuple-pairs for a tuple that is found to be strictly-dominated, and thus, avoids wasteful strict dominance checks.

As can be seen from Figures 5.26(a) and 5.27(a), the overhead in building the domain weighted versions of the index structures is negligible compared to the time taken in constructing the corresponding unweighted versions (Figure 5.16(a)). Also, like before, the index structure needs to be built only once and can be reused for different domain weighted Strata-Skyline (SS) queries. Therefore, the overhead in constructing the index structures can be further amortized over time.

Figure 5.26(b) shows query execution times on data that is not sorted. As can be observed, SS-WB [No RLE] performs better than SS-WB in all cases. This is because the lengths of the runs obtained in the RLE-compressed index structures are very

short when the data is not sorted, and hence, the gains achieved is not large enough to overcome the overhead incurred by the RLE-compressed element-wise multiplication operations. In fact, the cost of element-wise multiplications on the compressed unsorted data is very high; this is confirmed by the fact that SS-BNL performs much better than the SS-WB algorithm on data that is not sorted. Therefore, in this scenario, it is better to use SS-WB [No RLE] that performs better than SS-BNL, except in the case where the data incompleteness is extremely low ($I = 1\%$).

Another important observation from Figure 5.26(b) is that domain weighted SS query execution on data that does not follow any particular sort order is significantly more expensive than carrying out the corresponding unweighted SS queries. Hence, in Figure 5.27(b), the performance of the domain weighted SS algorithms is studied on data that is sorted according to the order discussed in Section 5.6.1. As can be observed, in this scenario, the SS-WB algorithm performs extremely well in all cases and is multiple orders of time faster than all the other domain weighted alternatives. The main reason for this behaviour is that the sorted MovieLens dataset is highly compressible and so the lengths of the runs obtained in the RLE-compressed index structures are long. Therefore, in this case, the gains achieved by using RLE-compressed element-wise multiplication operations is very large, while the overhead incurred is negligible. Also, the performance of the domain weighted SS-WB algorithm is very efficient and comparable to the performance of its unweighted counterparts. As shown by Figure 5.27(b), SS-WB incurs only a slight overhead in computing domain weighted skyline strata compared to the SS-EBU and SS-IBU algorithms.

Figure 5.27(b) also includes the results for SS-BNL [Naive]. The SS-BNL [Naive] algorithm is a naive version of the domain weighted SS-BNL algorithm. Unlike SS-BNL, the SS-BNL [Naive] algorithm does not avoid generating all tuple-pairs for a tuple that is found to be strictly-dominated. This means that SS-BNL [Naive] enumer-

I	1%	12%	26%	38%
Obtained based on discrete domain distribution				
SS-WB [Discrete Domain Distribution] (ms)	689.6	664.4	667.8	670.8
Obtained based on continuous domain distribution				
SS-WB [Continuous Domain Distribution; $\sigma^2 = 4$] (ms)	690.6	687.4	687.4	635.0
SS-WB [Continuous Domain Distribution; $\sigma^2 = 1$] (ms)	691.0	682.6	687.4	634.4
Matlab Initialization (s)	1.17			

(a) Domain weighted bitmap structure construction time

I	1%	12%	26%	38%
SS-WB [Discrete Domain Distribution] (s)	2.62	3.22	2.30	2.18
SS-WB [Continuous Domain Distribution; $\sigma^2 = 4$] (s)	2.69	3.22	2.31	2.21
SS-WB [Continuous Domain Distribution; $\sigma^2 = 1$] (s)	2.69	3.24	2.39	2.27

(b) Query execution time

Figure 5.28: Effect of computing $\omega(X, Y)$ values based on the continuous domain distribution assumption (MovieLens dataset, Sorted data, $n = 69K$, $d = 4$)

ates all possible tuple-pairs even when it executes strict dominance checks. As can be observed, SS-BNL performs significantly better than SS-BNL [Naive], and hence, SS-BNL is more efficient and optimized compared to SS-BNL [Naive].

Evaluations based on the Continuous Domain Distribution Assumption

In this subsection, the $\omega(X, Y)$ values for SS-WB are evaluated based on the assumption that the distribution of the values in the domains of the skyline attributes is normal and $f(x)$ (Section 5.2.3) is a Gaussian function with a default mean of $\mu = 5$. The variance of the Gaussian function considered were $\sigma^2 = 4$ and $\sigma^2 = 1$. The function to compute the $\omega(X, Y)$ values is implemented in MATLAB and a Java class was created for it using MATLAB's Builder JA[9] so that it could be integrated into

[9] http://www.mathworks.com/products/javabuilder/

the Java implementation of the SS-WB algorithm. Builder JA encrypts the MATLAB function and generates a Java wrapper around it so that it behaves just like any other Java class.

In Figure 5.28, the performance of the SS-WB algorithm on both discrete and continuous domain distributions is studied on data that is sorted according to the order discussed in Section 5.6.1. Figure 5.28(a) compares the overhead in building the domain weighted index structures based on both discrete and continuous domain distribution assumptions. Since duplicate attribute values in a dataset lead to identical weights in the WB index structure, the $\omega(X, Y)$ values under the continuous domain distribution assumption are calculated using the MATLAB function only once for each distinct value in the dataset and these precomputed weights are reused in building the entire WB index structure. As a result, the time taken to construct the WB structure based on continuous domain distribution is similar to the time taken to build the WB index structure based on the discrete domain distribution assumption, except that it suffers from an added overhead of ~ 1.2 s due to MATLAB class initialization. However, MATLAB class initialization needs to be done only once and this can be reused to compute the $\omega(X, Y)$ weight values for different continuous domain distribution based WB index structures. Hence, the overhead in constructing the index structures under the continuous domain distribution assumption can be further amortized over time.

Figure 5.28(b) shows that the SS-WB algorithm based on the continuous domain distribution assumption has a similar performance to its discrete domain distribution counterpart. The main reason for this behaviour is that, once the appropriate SS-WB index structures have been constructed, the two versions of SS-WB follow the same exact logic to execute Strata-Skyline (SS) queries. Therefore, both versions of the SS-WB algorithm are very efficient and perform identically.

5.8.7 Discussion

Based on the above experimental analysis, it can be concluded that the `SS` algorithms that utilize the unweighted potential dominance definition are efficient on both sorted and unsorted incomplete datasets. Hence, these algorithms do not require the data to be sorted in any particular order, though sorting does lead to better compression and further speed-ups in compressed bitwise AND/OR operations. On the other hand, `SS` algorithms based on weighted potential dominance are efficient only if the incomplete datasets are sorted and are highly compressible in nature. The main reason for this behaviour is that the element-wise multiplication operations on RLE-compressed index structures run significantly faster in this scenario.

Also, the proposed `SS` algorithms perform multiple orders of time faster than the `SS-BNL` algorithm. This is because the proposed algorithms leverage compressed domain operations during the stratification process. Whereas, `SS-BNL`, though optimized to avoid wasteful strict dominance checks, carries out the skyline stratification process through expensive tuple-to-tuple comparisons.

Chapter 6

CONCLUSIONS

In this dissertation, it was identified that a particular shortcoming of many of the existing skyline algorithms is that they primarily focus on single-source skyline processing in which all required skyline attributes are present in the same source. In other words, these algorithms commonly make an assumption that a skyline query is applied to a single static data source or data stream. However, this assumption does not hold true in many applications that require integration of data from different sources. In such scenarios, a skyline query may involve attributes belonging to multiple data sources, thus making the *join* operation an integral part of the overall process. Recently, various *skyline-join* algorithms have been proposed to address this problem in the context of static data sources. However, these algorithms suffer from several drawbacks: they often need to scan the data sources exhaustively to obtain the skyline-join results; moreover, the pruning techniques employed to eliminate tuples are largely based on expensive tuple-to-tuple comparisons. On the other hand, most data stream techniques focus on single stream skyline queries, thus rendering them unsuitable for *skyline-join* queries.

This dissertation also called attention to the fact that most of the earlier skyline algorithms count on the data being precise. In particular, these techniques typically make an assumption that the data is complete and all skyline attribute values are available. However, this assumption is not valid in many applications that involve incomplete data sources in which, due to reasons like data-entry errors and privacy, some of the attribute values are missing and are represented by NULL values. There exists a definition of dominance for incomplete data, but this leads to undesirable con-

sequences such as *non-transitive* and *cyclic* dominance both of which are detrimental to skyline processing.

Based on the aforementioned motivations, the main objective of the research work presented in this dissertation was the design and development of a framework of skyline operators that effectively handles three distinct types of skyline queries: 1) *skyline-join* queries on static data sources, 2) *skyline-window-join* queries over data streams, and 3) *strata-skyline* queries on incomplete datasets. This dissertation presented the unique challenges posed by these skyline queries and successfully addressed the shortcomings of current skyline techniques by proposing efficient methods to tackle the added overhead in processing skyline queries on static data sources, data streams, and incomplete datasets.

6.1 Skyline-Joins on Static Data Sources

In chapter 3, we studied the problem of processing skyline queries over multiple data sources. We proposed two novel non-iterative algorithms, namely $\mathtt{S^2J}$ and $\mathtt{S^3J}$, to process join-based skyline queries in a *skyline-sensitive* manner over two data sources. Both of these algorithms produce the skyline points by scanning the outer table one dominance layer at a time and require *at most* a single scan. A *trie*-based book-keeping strategy helps prune the tuples in the inner table, which are mapped to their corresponding Z-order values, quickly. The Z-order values help cluster the data into region-blocks in order to support efficient dominance checks and to facilitate block-based pruning of the inner table.

The $\mathtt{S^2J}$ algorithm scans the outer table entirely, while pruning the inner table progressively. The $\mathtt{S^3J}$ algorithm is similar to $\mathtt{S^2J}$, the main difference being that $\mathtt{S^3J}$ repeatedly swaps the outer and inner tables for symmetric operation. A special stopping condition, applicable when using this symmetric strategy, helps the algo-

rithm stop earlier than S^{2}J, without having to scan any of the input datasets in its entirety. The experiments carried out show the superiority in performance of the S^{2}J and S^{3}J algorithms on both real and synthetic datasets with various distributions, cardinalities and dimensions. The proposed algorithms are very efficient in executing skyline-join queries over two data sources and significantly outperform the existing skyline-sensitive join techniques.

The work presented in this chapter also extended S^{2}J's and S^{3}J's two-way skyline-join ability to process skyline-join queries over more than two data sources. We proposed the S^{2}J-M and S^{3}J-M algorithms that efficiently handle skyline-join queries over M data sources. We presented an extensive experimental analysis of the S^{2}J-M and S^{3}J-M algorithms. Based on these experimental results, we made recommendations on how to pick good *skyline-sensitive join* query plans that may potentially lead to the most reduction in wasted work during the processing of skyline-join queries over more than two data sources.

6.2 Skyline-Window-Joins on Data Streams

In the work presented in Chapter 4, we introduced and studied the problem of computing skyline-window-join (SWJ) queries over pairs of data streams. Recognizing that overlaps exist not only at the consecutive windows of the input data steams, but also consecutive windows of the individual iteration layers of skyline-computation, we first proposed a novel *iteration-fabric* processing structure to identify and eliminate per-layer overlaps across consecutive windows and then presented a Layered Skyline-window-Join (LSJ) operator that (a) partitions the overall process into processing layers and (b) maintains skyline-join results in an incremental manner by continuously monitoring the changes in all layers of the process. Extensive experimental evaluations over real and simulated data sets showed that the proposed approach

provides significant gains over processing schemes that are not designed to eliminate redundant work across multiple processing layers.

6.3 Strata-Skylines on Incomplete Datasets

In Chapter 5, we introduced the Strata-Skyline (SS) operator that efficiently handles skyline queries over incomplete data. We proposed two new definitions of dominance, namely *unweighted* and *domain weighted potential dominance*, to help identify *potential dominance relations* between tuples. The novel SS operator utilizes these definitions to stratify the tuples in an incomplete dataset into *strata* or *layers* of varying degrees of *skyline potential*. We developed two algorithms based on the unweighted potential dominance definition, namely Strata-Skyline-using-Explicit-Bitmaps-for-Unknowns (SS-EBU) and Strata-Skyline-using-Implicit-Bitmaps-for-Unknowns (SS-IBU), that are leveraged by the SS operator to carry out the stratification process. The SS-EBU algorithm utilizes the Explicit-Bitmaps-for-Unknowns (EBU) index, whereas SS-IBU exploits the Implicit-Bitmaps-for-Unknowns (IBU) index to efficiently answer SS queries by using fast bitwise operations over compressed bitmap structures. We also developed the Strata-Skyline-using-Weighted-Bitmaps (SS-WB) algorithm that is based on the domain weighted potential dominance definition. SS-WB utilizes the Weighted-Bitmaps (WB) index structure to effectively answer domain weighted SS queries by leveraging efficient compressed domain element-wise multiplication operations over RLE-compressed WB structures.

The Strata-Skyline (SS) operator does not ignore or replace unknown values, instead it incorporates their skyline potential into the core skyline computation process. We reported extensive experimental evaluations that confirm the advantages of the SS operator over extensions of existing schemes which are not designed to handle the added overhead in processing skyline queries over incomplete data.

REFERENCES

[1] Afshani, P., P. K. Agarwal, L. Arge, K. G. Larsen and J. M. Phillips, "(Approximate) uncertain skylines", in "ICDT", pp. 186–196 (2011).

[2] Alwan, A., H. Ibrahim, N. I. Udzir and S. Fatimah, "Estimating missing values of skylines in incomplete database", DEIS pp. 220–229 (2013).

[3] Atallah, M. J. and Y. Qi, "Computing all skyline probabilities for uncertain data", in "PODS", pp. 279–287 (2009).

[4] Avnur, R. and J. M. Hellerstein, "Eddies: Continuously adaptive query processing", in "In SIGMOD", pp. 261–272 (2000).

[5] Babcock, B., S. Babu, M. Datar, R. Motwani and J. Widom, "Models and issues in data stream systems", in "PODS", pp. 1–16 (2002).

[6] Babcock, B., M. Datar and R. Motwani, "Load shedding for aggregation queries over data streams", Data Engineering, International Conference on **0**, 350 (2004).

[7] Balke, W.-T., U. Güntzer and J. X. Zheng, "Efficient distributed Skylining for web information systems", in "EDBT", (2004).

[8] Bartolini, I., P. Ciaccia, V. Oria and M. T. Özsu, "Flexible integration of multimedia sub-queries with qualitative preferences", Multimedia Tools Appl. **33**, 3, 275–300 (2007).

[9] Bartolini, I., P. Ciaccia and M. Patella, "Salsa: computing the skyline without scanning the whole sky", in "CIKM", (2006).

[10] Bartolini, I., P. Ciaccia and M. Patella, "Efficient sort-based skyline evaluation", ACM Trans. Database Syst. **33**, 4, 31:1–31:49 (2008).

[11] Bartolini, I., P. Ciaccia and M. Patella, "Domination in the probabilistic world: Computing skylines for arbitrary correlations and ranking semantics", ACM Trans. Database Syst. **39**, 2, 14:1–14:45 (2014).

[12] Bhattacharya, A. and B. P. Teja, "Aggregate skyline join queries: Skylines with aggregate operations over multiple relations", in "COMAD", (2010).

[13] Börzsönyi, S., D. Kossmann and K. Stocker, "The Skyline operator", in "ICDE", pp. 421–430 (2001).

[14] Catania, B., G. Guerrini, M. T. Pinto and P. Podesta, "Relaxed queries over data streams", in "SEBD", (2012).

[15] Chomicki, J., P. Godfrey, J. Gryz and D. Liang, "Skyline with presorting", in "ICDE", (2003).

[16] Cortes, C., K. Fisher, D. Pregibon, A. Rogers and F. Smith, "Hancock: A language for extracting signatures from data streams", in "SIGKDD", pp. 9–17 (2000).

[17] Cranor, C., Y. Gao, T. Johnson, V. Shkapenyuk and O. Spatscheck, "Gigascope: High performance network monitoring with an sql interface", in "ACM SIGMOD", (2002).

[18] Das, A., J. Gehrke and M. Riedewald, "Approximate join processing over data streams", in "Proceedings of the 2003 ACM SIGMOD international conference on Management of data", pp. 40–51 (2003).

[19] Das Sarma, A., A. Lall, D. Nanongkai and J. Xu, "Randomized multi-pass streaming skyline algorithms", VLDB **2**, 85–96 (2009).

[20] Ding, L., N. Mehta, E. A. Rundensteiner and G. T. Heineman, "Joining punctuated streams", in "EDBT", pp. 587–604 (2004).

[21] Endres, M., P. Roocks, F. Wenzel, A. Huhn and W. Kießling, "Handling of null values in preference database queries", ECAI (2012).

[22] Fagin, R., "Combining fuzzy information from multiple systems", in "PODS", pp. 216–226 (ACM Press, 1996).

[23] Fagin, R., A. Lotem and M. Naor, "Optimal aggregation algorithms for middleware", in "PODS", pp. 102–113 (2001).

[24] Golab, L. and M. T. Özsu, "Processing sliding window multi-joins in continuous queries over data streams", in "VLDB", pp. 500–511 (2003).

[25] Haas, P. J. and J. M. Hellerstein, "Ripple joins for online aggregation", in "SIGMOD", pp. 287–298 (1999).

[26] Hose, K. and A. Vlachou, "Distributed skyline processing: A trend in database research still going strong", in "Proceedings of the 15th International Conference on Extending Database Technology", EDBT '12, pp. 558–561 (ACM, 2012).

[27] Hose, K. and A. Vlachou, "A survey of skyline processing in highly distributed environments", The VLDB Journal **21**, 3, 359–384 (2012).

[28] Huang, J., J. Chen, Q. Du and J. Yin, "A load balancing skyline query algorithm in high bandwidth distributed systems", in "FSKD", (2010).

[29] Ilyas, I. F., W. G. Aref and A. K. Elmagarmid, "Supporting top-k join queries in relational databases", in "VLDB", pp. 754–765 (2003).

[30] Ilyas, I. F., G. Beskales and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems", ACM Comput. Surv. **40**, 4, 11:1–11:58 (2008).

[31] Jiang, B. and J. Pei, "Online interval skyline queries on time series", in "ICDE", pp. 1036–1047 (2009).

[32] Jin, W., M. Ester, Z. Hu and J. Han, "The multi-relational skyline operator", ICDE (2007).

[33] Jin, W., M. D. Morse, J. M. Patel, M. Ester and Z. Hu, "Evaluating skylines in the presence of equijoins", in "ICDE", pp. 249–260 (2010).

[34] Khalefa, M. E., M. F. Mokbel and J. J. Levandoski, "Skyline query processing for incomplete data", in "ICDE", pp. 556–565 (2008).

[35] Khalefa, M. E., M. F. Mokbel and J. J. Levandoski, "Skyline query processing for uncertain data", in "CIKM", pp. 1293–1296 (2010).

[36] Khalefa, M. E., M. F. Mokbel and J. J. Levandoski, "Prefjoin: An efficient preference-aware join operator", in "ICDE", pp. 995–1006 (2011).

[37] Kossmann, D., F. Ramsak and S. Rost, "Shooting stars in the sky: An online algorithm for Skyline queries", in "VLDB", (2002).

[38] Kung, H.-T., F. Luccio and F. P. Preparata, "On finding the maxima of a set of vectors", J. ACM (1975).

[39] Lee, K. C. K., B. Zheng, H. Li and W.-C. Lee, "Approaching the skyline in Z order", in "VLDB", (2007).

[40] Lemire, D., O. Kaser and K. Aouiche, "Sorting improves word-aligned bitmap indexes", Data Knowl. Eng. **69**, 1, 3–28 (2010).

[41] Levandoski, J. J., M. F. Mokbel and M. E. Khalefa, "Flexpref: A framework for extensible preference evaluation in database systems", in "ICDE", (2010).

[42] Li, H.-G., S. Chen, J. Tatemura, D. Agrawal, K. S. Candan and W.-P. Hsiung, "Safety guarantee of continuous join queries over punctuated data streams", in "VLDB", pp. 19–30 (2006).

[43] Li, X., Y. Wang, X. Li and G. Wang, "Skyline query processing on interval uncertain data", in "ISORCW", pp. 87–92 (2012).

[44] Lin, X., Y. Yuan, W. Wang and H. Lu, "Stabbing the sky: Efficient skyline computation over sliding windows", in "ICDE", (2005).

[45] Liu, X., D.-N. Yang, M. Ye and W.-C. Lee, "U-skyline: A new skyline query for uncertain databases", TKDE **25**, 4, 945–960 (2013).

[46] Lofi, C., K. E. Maarry and W.-T. Balke, "Skyline queries over incomplete data - error models for focused crowd-sourcing", in "ER", (2013).

[47] M. Morton, G., "A computer oriented geodetic data base; and a new technique in file sequencing", Tech. rep. (1966).

[48] Madden, S. and M. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data", in "ICDE", (2002).

[49] Marian, A., N. Bruno and L. Gravano, "Evaluating top-k queries over web-accessible databases", ACM Trans. on Database Syst. **29**, 2, 319–362 (2004).

[50] Matoušek, J., "Computing dominances in E^n", Information Processing Letters **38**, 277–278 (1991).

[51] Mouratidis, K., S. Bakiras and D. Papadias, "Continuous monitoring of top-k queries over sliding windows", in "SIGMOD", pp. 635–646 (2006).

[52] Nagendra, M. and K. S. Candan, "Skyline-sensitive joins with LR-pruning", in "EDBT", pp. 252–263 (2012).

[53] Natsev, A., Y.-C. Chang, J. R. Smith, C.-S. Li and J. S. Vitter, "Supporting incremental join queries on ranked inputs", in "VLDB", pp. 281–290 (2001).

[54] Nehme, R. V. and E. A. Rundensteiner, "Scuba: Scalable cluster-based algorithm for evaluating continuous spatio-temporal queries on moving objects", in "EDBT", pp. 1001–1019 (2006).

[55] Papadias, D., Y. Tao, G. Fu and B. Seeger, "Progressive Skyline computation in database systems", ACM (2005).

[56] Park, N. H., V. Raghavan and E. A. Rundensteiner, "Supporting multi-criteria decision support queries over time-interval data streams", in "DEXA: Part I", pp. 281–289 (2010).

[57] Pei, J., B. Jiang, X. Lin and Y. Yuan, "Probabilistic skylines on uncertain data", in "VLDB", pp. 15–26 (2007).

[58] Peng, L., R. Yu, K. S. Candan and X. Wang, "Object and combination shedding schemes for adaptive media workflow execution", IEEE TKDE **22**, 1, 105–119 (2010).

[59] Raghavan, V. and E. A. Rundensteiner, "Progressive result generation for multi-criteria decision support queries", in "ICDE", pp. 733–744 (2010).

[60] Raghavan, V., E. A. Rundensteiner and S. Srivastava, "Skyline and mapping aware join query evaluation", Inf. Syst. **36**, 917–936 (2011).

[61] Sahpaski, D., A. S. Dimovski, G. Velinov and M. Kon-Popovska, "Efficient processing of top-k join queries by attribute domain refinement", in "ADBIS", pp. 318–331 (2012).

[62] Shang, H. and M. Kitsuregawa, "Skyline operator on anti-correlated distributions", PVLDB **6**, 9, 649–660 (2013).

[63] Shastri, A., D. Yang, E. A. Rundensteiner and M. O. Ward, "Mtops: Scalable processing of continuous top-k multi-query workloads", in "CIKM", pp. 1107–1116 (2011).

[64] Srivastava, U. and J. Widom, "Memory-limited execution of windowed stream joins", in "VLDB", pp. 324–335 (2004).

[65] Steuer, R. E., *Multiple Criteria Optimization: Theory, Computation and Application* (John Wiley, New York, 546 pp, 1986).

[66] Stockinger, K. and K. Wu, "Bitmap indices for data warehouses", in "In Data Warehouses and OLAP. 2007. IRM", (2006).

[67] Stojmenović, I. and M. Miyakawa, "An optimal parallel algorithm for solving the maximal elements problem in the plane", Parallel Computing **7**, 249–251 (1988).

[68] Sun, D., S. Wu, J. Li and A. K. H. Tung, "Skyline-join in distributed databases", in "ICDE Workshops", (2008).

[69] Sun, S., Z. Huang, H. Zhong, D. Dai, H. Liu and J. Li, "Efficient monitoring of skyline queries over distributed data streams", Knowl. Inf. Syst. **25**, 575–606 (2010).

[70] Tan, K.-L., P.-K. Eng and B. C. Ooi, "Efficient progressive Skyline computation", in "VLDB", (2001).

[71] Tao, Y. and D. Papadias, "Maintaining sliding window skylines on data streams", IEEE TKDE **18**, 377–391 (2006).

[72] Tatbul, N., U. Çetintemel, S. Zdonik, M. Cherniack and M. Stonebraker, "Load shedding in a data stream manager", in "VLDB", pp. 309–320 (2003).

[73] Tatbul, N. and S. Zdonik, "Window-aware load shedding for aggregation queries over data streams", in "VLDB", pp. 799–810 (2006).

[74] Trimponias, G., I. Bartolini, D. Papadias and Y. Yang, "Skyline processing on distributed vertical decompositions", IEEE Transactions on Knowledge and Data Engineering **25**, 4, 850–862 (2013).

[75] Tucker, P. A., D. Maier, T. Sheard and L. Fegaras, "Exploiting punctuation semantics in continuous data streams", IEEE TKDE (2003).

[76] Viglas, S. D., J. F. Naughton and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources", in "VLDB", pp. 285–296 (2003).

[77] Vlachou, A., C. Doulkeridis and Y. Kotidis, "Angle-based space partitioning for efficient parallel Skyline computation", in "SIGMOD", (2008).

[78] Vlachou, A., C. Doulkeridis and N. Polyzotis, "Skyline query processing over joins", in "SIGMOD", (2011).

[79] Wang, S., E. A. Rundensteiner, S. Ganguly and S. Bhatnagar, "State-slice: New paradigm of multi-query optimization of window-based stream queries", in "VLDB", pp. 619–630 (2006).

[80] Wang, X., K. S. Candan and J. Song, "Complex pattern ranking (CPR): Evaluating top-k pattern queries over event streams", in "DEBS", (2011).

[81] Wilschut, A. N. and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment", in "PDIS", pp. 68–77 (1991).

[82] Wu, K., E. J. Otoo and A. Shoshani, "Optimizing bitmap indices with efficient compression", ACM TODS **31**, 1, 1–38 (2006).

[83] Wu, M., L. Berti-Equille, A. Marian, C. M. Procopiuc and D. Srivastava, "Processing top-k join queries", PVLDB **3**, 1, 860–870 (2010).

[84] Wu, P., C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal and A. E. Abbadi, "Parallelizing Skyline queries for scalable distribution", in "EDBT", pp. 112–130 (2006).

[85] Xie, J., J. Yang and Y. Chen, "On joining and caching stochastic streams", in "ACM SIGMOD", pp. 359–370 (2005).

[86] Xu, P. and S. Tirthapura, "Optimality of clustering properties of space-filling curves", ACM Trans. Database Syst. **39**, 2, 10:1–10:27 (2014).

[87] Zhang, W., X. Lin, Y. Zhang, W. Wang and J. X. Yu, "Probabilistic skyline operator over sliding windows", in "ICDE", pp. 1060–1071 (2009).